Yijun Yu · Arosha Bandara · Shinichi Honiden
Zhenjiang Hu · Tetsuo Tamai · Hausi Muller
John Mylopoulos · Bashar Nuseibeh *Eds.*

Engineering Adaptive Software Systems

Communications of NII Shonan Meetings

Springer

Engineering Adaptive Software Systems

Yijun Yu • Arosha Bandara • Shinichi Honiden
Zhenjiang Hu • Tetsuo Tamai • Hausi Muller
John Mylopoulos • Bashar Nuseibeh
Editors

Engineering Adaptive Software Systems

Communications of NII Shonan Meetings

Springer

Editors
Yijun Yu
The Open University
Milton Keynes, UK

Arosha Bandara
The Open University
Milton Keynes, UK

Shinichi Honiden
National Institute of Informatics
Tokyo, Japan

Zhenjiang Hu
National Institute of Informatics
Tokyo, Japan

Tetsuo Tamai
Hosei University
Tokyo, Japan

Hausi Muller
University of Victoria
Victoria, BC, Canada

John Mylopoulos
University of Toronto
Toronto, Canada

Bashar Nuseibeh
The Open University
Milton Keynes, UK

ISBN 978-981-13-2184-9 ISBN 978-981-13-2185-6 (eBook)
https://doi.org/10.1007/978-981-13-2185-6

Library of Congress Control Number: 2018961997

This Springer imprint is published by the registered company Springer Nature Singapore Pte Ltd.
The registered company address is: 152 Beach Road, #21-01/04 Gateway East, Singapore 189721, Singapore

Preface

The first Shonan Meeting on Engineering Adaptive Systems (EASy) [1], which was held in 2012, generated heated discussions on the problems and challenges about self-managing systems. Participants from multiple disciplinaries reached the consent that EASy has by no means an easy solution in software engineering alone, not to mention many other challenges in general system engineering.

The organisers of the following Shonan meetings [2, 3] decided to focus on the problems and solutions that can help engineer adaptive software, hence a change of the focus to Engineering Adaptive Software Systems (EASSy). The technical reports above have gathered from abstracts of all individual participants; however, there has not yet been a full report on the crux of interesting viewpoints, which could collaboratively pave the way to solve some aspects of the long-standing research problems.

This book is a collection of materialised reflections by some of our active participants present in much greater details, which we hope can fuel a tank of thoughts for engineering the next-generation adaptive software systems.

The chapters included in the book have a good coverage of the area, ranging from design and engineering principles (Chap. 1) to control-theoretic solutions (Chap. 2) and bidirectional transformations (Chap. 3), which can be seen as promising ways to implement the functional requirements of self-adaptive systems. Important quality requirements are also dealt with by these approaches: parallel adaptation for performance (Chap. 4), self-adaptive authorization infrastructure for security (Chap. 5), and self-adaptive risk assessment for self-protection (Chap. 6). Finally, Chap. 7 provides a concrete self-adaptive robotics operating system as a testbed for self-adaptive systems.

Although by no means a complete coverage of all possible research topics, these chapters can be seen as concrete research agenda's proposed by experts in these areas.

In a nutshell, we hope the book will initiate promising progresses in this interdisciplinary research field.

Shonan Village, Japan, EASSy Shonan Meetings Organisers
Milton Keynes, UK Yijun Yu
Milton Keynes, UK Arosha Bandara
Tokyo, Japan Shinichi Honiden
Tokyo, Japan Zhenjiang Hu
Tokyo, Japan Tetsuo Tamai
Victoria, BC, Canada Hausi Muller
Toronto, Canada John Mylopoulos
Milton Keynes, UK Bashar Nuseibeh
July 2018

References

1. Bandara, A., Yu, Y., Nuseibeh, B., Honiden, S.: Engineering adaptive systems. In: Shonan Meetings, Shonan Technical Report 003, Shonan, Japan (2012)
2. Yu, Y., Honiden, S., Muller, H.A., Mylopoulos, J.: Engineering adaptive software systems. In: Shonan Meetings, Shonan Technical Report 027, Shonan, Japan (2013)
3. Tamai, T., Muller, H.A., Nuseibeh, B.: Engineering adaptive software systems. In: Shonan Meetings, Shonan Technical Report 052, Shonan, Japan (2015)

Contents

Chapter 1
Design and Engineering of Adaptive Software Systems

Soichiro Hidaka, Zhenjiang Hu, Marin Litoiu, Lin Liu, Patrick Martin, Xin Peng, Guiling Wang, and Yijun Yu

Abstract New challenges such as big data, ultra-large-scale services, and continuously available services are driving the evolution to adaptive software systems, which are able to modify their behavior in response to their environmental and

This chapter was edited by Soichiro Hidaka and Patrick Martin.

S. Hidaka (✉)
Hosei University, Tokyo, Japan
e-mail: hidaka@hosei.ac.jp

Z. Hu
National Institute of Informatics, Tokyo, Japan
e-mail: hu@nii.ac.jp

M. Litoiu
York University, Toronto, ON, Canada
e-mail: mlitoiu@yorku.ca

L. Liu
School of Software, Tsinghua University, Beijing, China
e-mail: linliu@tsinghua.edu.cn

P. Martin
Queen's University, Kingston, ON, Canada
e-mail: martin@cs.queensu.ca

X. Peng
Fudan University, Shanghai, China
e-mail: pengxin@fudan.edu.cn

G. Wang
Beijing Key Laboratory on Integration and Analysis of Large-Scale Stream Data,
North China University of Technology, Beijing, China
e-mail: wangguiling@ict.ac.cn

Y. Yu
The Open University, Milton Keynes, UK
e-mail: y.yu@open.ac.uk

Y. Yu et al. (eds.), *Engineering Adaptive Software Systems*,
https://doi.org/10.1007/978-981-13-2185-6_1

internal changes, in order to achieve their goals. Providing support in all phases of the life cycle of adaptive software systems is thus an important challenge facing the software engineering research community. This chapter highlights current research on methods and techniques for the design and engineering of adaptive software systems. The design space for self-adaptive systems is first examined, and then a goal-oriented framework for adaptive service composition is described. The human factors component of the design of adaptive systems are considered from four different points of view. We then argue that model management from database community can be adapted for effective development of self-adaptive systems. Finally, sustainability of adaptive components are shown to be achieved by making requirements future-proof.

1.1 Introduction

The need for software systems to accommodate the demands of new challenges such as big data, ultra-large-scale services, and continuously available services is driving the evolution to adaptive software systems, which are able to modify their behavior in response to changes in their external environment, or in the system itself, in order to achieve their goals. The mechanisms to achieve these adaptations must therefore be one of the focuses of the system's development and maintenance.

There is a strong agreement in the self-adaptive systems community that the design and implementation of self-adaptive systems should be based on feedback loops. Brun et al. [9], in their earlier research roadmap paper, emphasize feedback loops as first-class design entities and argue that loops are the essential features in controlling and managing uncertainties in software systems. They assert that, by making feedback loops visible, the impact of these feedback loops on system behavior could be identified earlier and the most important properties of self-adaptation could be addressed. Feedback loops are therefore an important focus of the work discussed in the chapter.

Providing support in all phases of the life cycle of adaptive software systems is an important challenge facing the software engineering research community [15, 20]. This chapter highlights current research on methods and techniques for the design and engineering of adaptive software systems.

1.2 Designing Adaptive Software Systems

The design of adaptive software systems involves making decisions about observing the environment and the system itself, selecting adaptation mechanisms and enacting the mechanisms [20]. In this section Litoiu first examines the design space for adaptive systems on Software Defined Infrastructure (SDIs) such as clouds. SDIs introduce additional complexity for the design of adaptive systems in that an adaptation performed by a system can affect both the system and the SDI.

Service orientation is a leading form for Web-based software applications that allow distributed enterprise business processes to integrate through the composition of individual independently developed services. Composite services, however, need to evolve and reorganize as new requirements and services emerge. Liu and Wang next describe a goal-oriented framework for adaptive service composition.

Human factors are another important aspect of design since humans influence the adaptive software systems in various ways. Adaptive software systems fulfill the stakeholders' requirements through the cooperation of human, hardware, and software. Peng considers the impact of human involvement on the design of adaptive software systems from four different viewpoints, namely, experts, users, agents, and components.

1.2.1 Design Space for Self-Adaptive Systems

Designing an adaptive system encompasses decisions such as what and how to monitor the system and its environment, how and when to select and activate adaptations, etc. More recently, Brun et al. [10] introduce the concept of design space for adaptive systems, which contains key questions to ask when designing a self-adaptive system. The authors present a conceptual model of how to identify different components of an adaptive system by answering a set of questions along five dimensions: identification, observation, representation, control, and adaptation mechanisms.

In recent years, companies and institutions have started to deploy and run their applications on Software Defined Infrastructure (SDI), typified by clouds. The main consequence of this is that the self-adaptive feedback loops built around an application can change not only the application but the application's environment, that is, the cloud, as well. One example of such a SDI is SAVI cloud[1] which is an Extended Cloud consisting of *edges* and a *core*. The edges are resource-constrained clouds, primarily aimed to host latency-sensitive components/services of the application. Meanwhile, the core has ample resources and is capable of executing the more resource-intensive components/services.

Feedback loops in a SDI have to consider the particularities of the infrastructure, especially the fact that the SDI is programmable and resources (software services, network, storage), exposed as services, can be acquired, released, or tuned at runtime. Figure 1.1 shows a reference feedback loop for applications deployed on a SDI. There are two basic complementary sets of loops, composed of varied services, that can work at different time scales, with different performance:

- Reactive feedback (black arrows): reacts to current load and events, implements simple control decisions such as PID (proportional, integrative, derivative). These loops are fast but are limited in the adaptation solutions they provide

Contributed by Marin Litoiu.

[1] http://www.savinetwork.ca

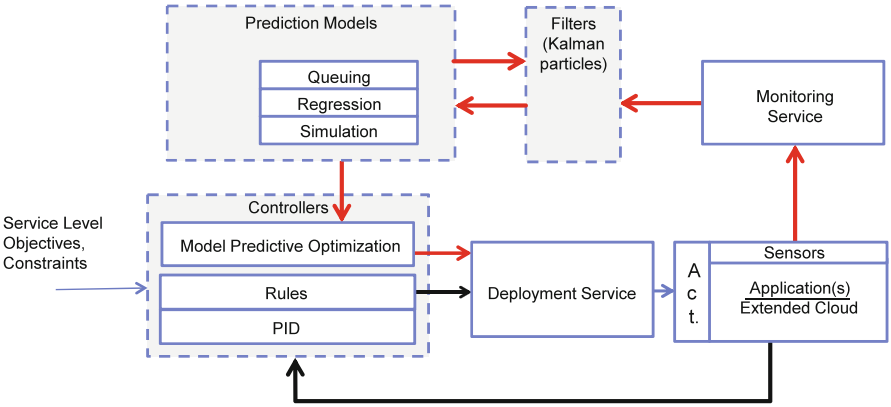

Fig. 1.1 Adaptive feedback loops for applications deployed in cloud

- Predictive feedback (red arrows): anticipates future load and events and can consider more complex decisions. These loops can use prediction models, filters, and predictive optimization. They are slow and effortful but produce effective results

Next we describe the elements of the predictive feedback loops and then discuss the design decision space for these loops.

Monitoring is done through a special Monitoring service which has to fuse metrics from both application and SDI. To build a model of the application (performance, cost, security, etc.), the feedback loop might need to estimate data and metrics which are not directly accessible by the Monitoring service, hence, the need to have Filters or Estimator services (Kalman and Particle filters are example of estimators). The Prediction Models service uses the data measured or estimated to predict future states of the system. At the same time, the predicted data is fed back into the Filters to adapt their estimation. Controllers are the decision services of the feedback loop. Based on the current and estimated states and based on the Service Level Objectives and Constraints, Controllers decide what adaptation needs to take place. These adaptations are implemented by a Deployment Service which through a set of APIs (Act. in Fig. 1.1) has access to both the application and SDI control parameters (or control points).

The feedback loops represented in Fig. 1.1 follow the MAPE-K architecture. For example, Prediction Models represent the Knowledge (K), while Controllers represent the Analysis (A), etc.

Next, we present decisions to be considered when designing feedback loops. We group them in three clusters, similar to the approach in Brun et al. [10], but augmented with specific questions for these new types of deployments.

Observation cluster includes questions related to the design of Monitoring and the Filter services. The designer has to answer questions about what is monitored, what is the source of the data (application, SDI), when and how often monitoring

occurs, and what is the format of data. For the Filter service, the designer has to answer what and how states are determined based on the monitored data and what types of filters are to be considered (besides Kalman and Particle filters).

Representation cluster relates to the Prediction Models service. It is concerned with the runtime representation of the quantitative dependencies among adaptation targets (SLO), perturbations, Actuator variables (or control points), measured metrics, and internal states. Models capture these representations, and they can include Queuing, Regression, and Simulation formalisms, but the spectrum of models one can consider is quite wide. What type of model to consider is the main dimension of this cluster. It might be the case that one model is not enough to capture the QoS under consideration. For example, Queuing models can capture steady states, while Simulation or Regression models can capture transients and nonstationary conditions. When and how to switch between the models are design questions that depend on the application, workloads, and operating conditions and define another dimension of this decision cluster.

Control cluster is concerned with the mechanisms through which an adaptation solution is enacted through the Controllers service. This can be a very rich cluster because it explores the space of possible adaptation solutions and the mechanisms to find those solutions. Optimization can be the mechanism that connects the current state with the desired states and goals. Optimization has been explored in [36] and [35] as a way to achieve a performance goal while optimizing the cost. When the goal has to be achieved in the future, an optimization has to optimize the sequence of decisions over a finite time horizon, a technique called Model Predictive Control [26]. Defining an objective function to optimize can be a laborious process. Zoghi et al. [59] showed a methodology that constructs an objective function based on the adaptation goals and the control points in the system. The control points are the APIs available at the Actuators, and they can belong to application, to cloud, or to multi-cloud. In general, the optimization algorithms use the Prediction Models to estimate and predict the effect of different decisions. It might be the case that, since there are many models used by a feedback loop, there should be many controllers as well. Switching between them can become a complex task that might require a higher-layer feedback loop that supervises the performance of different models and optimizers.

1.2.2 Design of Adaptive Services

In the services adaptation setting, in order to effectively accommodate changes and provide satisfactory services to customers, a provider has to know what services to provide, with what functional features, and at what quality level. To this end, a general conceptual model of services needs to be in place to support the adap-

*Contributed by Lin Liu and Guiling Wang.

tation process. Service orientation as a form for Web-based software applications allows distributed enterprise business processes to integrate through composition of individual services developed independently. Composability provides desirable flexibility and reusability in building distributed enterprise information systems. However, existing composite services need to evolve and reorganize as new requirements and services emerge.

While many techniques of conventional requirements elicitation could potentially be applied to services, their open and dynamic nature introduces new challenges.

There are existing technical solutions – such as middleware adapters, software agents, and adaptive architectures – intended to link software components through various kinds of interaction styles. However, the ability to respond to changes in the environment by means of reorganization, thereby exhibiting context awareness and adaptability, has yet to be supported by any existing runtime platform.

Many studies in the requirements engineering (RE) literature suggest the importance of runtime requirements monitoring and adaptation. In [24], Fickas and Feather have clearly illustrated the significance of monitoring requirements based on the analysis of two commercial case studies. A comprehensive probe into the research challenges in this area is provided in [3], which pointed out, among other things, that requirements engineering for self-adaptive systems must deal with uncertainty and treat requirements as runtime entities. Jureta et al. [31] propose a formalism for modeling dynamic requirements. All these efforts agree on the importance of the problem of requirements modeling, monitoring, and evolution, on an ongoing basis, for a running, live, and ever-changing system.

This chapter further develops the concepts and techniques of goal-oriented requirements analysis, to seek answers to the following questions: (1) how to model and refine the high-level requirements of service users at runtime and map them into abstract service composition architectures and (2) how to enable compositional adaptation through the maintenance and monitoring of such requirements models and their mapping to services structure.

Our proposed framework is founded on our previous work on a service ontology [37] and an analysis of goal-oriented modeling for runtime requirements [30, 39]. Using this research baseline, this chapter proposes service-oriented requirements monitoring and a continuous negotiation and adaptation process. Our proposal exploits existing goal-oriented service requirements models and new service requests to establish an alignment between service ends (defined by requirements) and service means (defined by the current pool of available services). Current environment variables being monitored include service availability, quality of service (QoS) status, user service needs, and operating environment variables.

We suggest that a service model consists of the following basic elements: services context, high-level composite services, service interface, and service implementation. Service is the core and major entity involved in the adaptation process. Context represents external factors influencing the system or influenced by the system, which in general is the major cause for changes and adaptations of services. In order to reduce the negative impact of change, the service composition and reconfiguration

is used to adapt the existing services. Services are composed of one or more elementary services, each of which has associated quality attributes. Two major processes are defined in our framework: adaptation and composition. Adaptation can be imposed on either the service interface or the service implementation, which are two predefined strategies to accommodate changing requests.

With regard to *Service*, this chapter takes the position that service systems should not be designed as to be able to predict how user requirements will change. The system design should be relatively simple and easy to maintain. For the self-adaptable service systems, the basic elements of a system should be fine-grained and ready for compositional adaptation. Service systems are composed of two major parts: functionalities and QoS. System functions are supported by services, and a service can provide one or more function points by structural composition. QoS relates to the nonfunctional properties of a system, such as cost, performance, reliability, security, and so on. Composite services also have QoS, which is essentially a function of the QoS attributes of the element services. If a system's QoS satisfies the constraints of the requesters, the system is suitable for the current context. Otherwise, the system has to take some actions to adapt itself according to the environment.

In the followings, we introduce the model of runtime requirements for adaptive service composition, followed by the adaptive service composition approach based on runtime requirements monitoring.

1.2.2.1 A Modeling Framework for Adaptive Service Composition

The basic elements of our proposed framework for adaptive service composition (ASC for short) consist of one or more service classes, represented with a service specification (e.g., DownloadDVD(x)). Each service specification has one or more instances that represent concrete services that implement the abstract service they are instances of. All service instances associated with a service class implement the same functionality, but the level of QoS may vary. In addition, an ASC model includes a set of service functionalities. Functionalities are associated with service classes and instances through Function. If S is an abstract service and s_i one of its instances, then SPECIFICATION$(s_1, \ldots, s_m) \models$ Function(S).

QoS for different services is represented through quality attributes, such as "*average response time*." Concrete services are assigned values for each quality attribute through the function Quality. If the average response time of service s is 1 s, *Quality*(*AverageResponseTime, s*) = 1 s. Quality attribute values can be considered as constraints on the range of a quality attribute. Services are maintained in a service repository. Due to the dynamic nature of the services environment, the service repository is changing constantly.

User requirements are represented as goals, using the notation of goal-oriented requirements engineering languages such as $i*$ [53] and Tropos [11] and earlier AI literature on planning [12]. Nonfunctional requirements, such as QoS, are represented as softgoals. For each softgoal, R, with an associated quality attribute

Table 1.1 Meaning of grammar rules

Rule	Meaning
seq($s+$)	Sequential execution of atomic services $s+$
loop(s, n)	Repeated execution of atomic service s, n times
sel($s+$)	Conditional selection of atomic services $s+$
par_and($s+$)	Concurrent execution of atomic services $s+$ (with complete synchronization)
par_or($s+$)	Concurrent execution of atomic services $s+$ (with 1 out of n synchronization)

Q and a constraint, such as "*responseTime* $<$ *2* s", a concrete service fulfills R iff $Quality(Q, s) \models Constraint(R)$, for example, the quality attribute value "*responseTime* $<$ 1sec" entails the constraint "*responseTime* $<$ 2 s."

We now assume that all the abstract services that correspond to a specification *Sp* have been composed in order to satisfy the goal entailed by *Sp*, and hence we can derive an abstract service composition structure based on the goal-refinement structure and the hidden temporal/casual constraints between all services in *Sp*. We use the following set of composition operators: $seq(s+)$, $loop(s, n)$, $sel(s+)$, $par\_and(s+)$, and $par\_or(s+)$, where s denotes an atomic service, $s+$ denotes a set of one or more services, and *seq*, *loop*, *sel*, *par_and*, *par_or* are introduced in Table 1.1.

A composite abstract service can be instantiated into a composite one by assigning an atomic concrete service to each atomic abstract service. Thus, there are two levels of composition. When composition is derived from a specification *Sp*, we take advantage of the goal model from which the specification was derived. In particular, we compose functionalities, where sequential composition ($seq(s+)$), conditional selection of composition ($sel(s+)$), and concurrent execution with complete synchronization ($par\_and(s+)$) are used. Sequential composition and concurrent execution are derived from and-decomposition of goals, while conditional selection of composition is derived from or-decomposition of goals.

Services are monitored to ensure that QoS requirements are met. This is accomplished by monitoring environmental variables, such as time and location, or specific variables, such as object internal states, and the application or system runtime performance variables, such as CPU utilization, memory usage, bandwidth availability, network stability, average user waiting time, or number of concurrent online users.

Services in S have associated quality constraints on the expected value or range thereof for a service quality (e.g., performance). The function Constraints map each service in S to a set of associated constraints. For example, if the service "*Provide Video Tutorials*" only requires "*Network speed above 50* kbps," then Constraints(*Provide Video Tutorials*) = {*Network speed* $\geq$ *50* kbps}.

Environmental variables influence both the status of services and user goals and their refinements. For example, if one of the constraints associated with service "*Provide Video Tutorials*" is "*Network speed* $>$ *50* kbps" and environmental variable

Table 1.2 Revoking action

Action	*Revoke*(s_{ij}, s_{il})
Precondition	Constraint(s_{il}) $\wedge$(bound(s_{ij}) $\wedge$ (Function(s_{il}) $\models$ Function(s_{ij})) $\wedge$Quality(s_{il}) $\models$ Quality(G) $\vee\neg$Available(s_{ij}))
Trigger	Quality(s_{il}) $\models$ Quality(s_{ij}) $\wedge$ available(s_{il})
Effect	bound(s_{il}) $\wedge$ remove(s_{ij})

Table 1.3 *Par_and*(s_{i1}, s_{ij}, s_{il}) Composition action

Action	*Par_and*(s_{i1}, s_{ij}, s_{il})
Precondition	Constraint(s_{i1}, s_{ij}, s_{il}) holds $\wedge$(Function(s_{i1}) $\models$ Function(s_{ij})) $\wedge$ Function(s_{i1}) $\models$ Function(s_{il})) $\wedge$minimal(throughput(G)) $\geq$ throughput(s_{i1}) $\wedge$minimal(throughput(G)) $\geq$ throughput(s_{ij}) $\wedge$minimal(throughput(G)) $\geq$ throughput(s_{il}) $\wedge$available(s_{i1}) $\wedge$ available(s_{ij}) $\wedge$ available(s_{il})
Trigger	throughput(Par_and(s_{i1}, s_{ij}, s_{il})) $\geq$ minimal(throughput(G))
Effect	bound(s_{i1}) $\wedge$ bound(s_{ij}) $\wedge$ bound(s_{il})

"*Network speed*" has a value that is less than 50 kbps, then the service is not available.

An adaptive system can perform actions to handle changes of user goals, the environment, or service components. Each action represents a specific procedure to be carried out by the system. Adaptation actions include adding, removing, or updating an atomic service; reorganizing the structure of an abstract service plan; reorganizing the structure of a concrete service chain and adding, removing, or updating user goals; and adding, removing, or updating an environmental monitor.

For each action there may be associated environmental conditions that need to be monitored by the system and included in the environment set E. Actions have associated preconditions, triggers, and effects with usual semantics. All of them consist of propositions constraining environmental variables and the internal state of a service component.

For example, when a new atomic service s_{il} emerges, s_{ij} is an atomic service of a composite service that suddenly becomes unavailable or waiting to be executed at the moment, i.e., Functionality(s_{il}) = Functionality(s_{ij}), and Quality(s_{il}) is obviously better than Quality (s_{ij}); the system may take "revoking" action in Table 1.2.

When a user's required value exceeds the QoS value of any existing service $e_1, \ldots, e_n$, the system may execute more than one atomic service (say s_{i1}, s_{ij}, s_{il}) concurrently (with complete synchronization) to meet the demand. This is shown as action "*Par_and*(s_{i1}, s_{ij}, s_{il}) *composition*" in Table 1.3.

Table 1.4 *Par_or*(s_{ij}, s_{il}) Composition action

Action	*Par_or*(s_{ij}, s_{il})
Precondition	Constraint(s_{ij},s_{il}) holds $\wedge$(Function(s_{ij}) == Function(s_{il})) $\wedge$minimal(reliability(G)) $\leq$ reliability(s_{ij}) $\leq$ optimal(reliability(G)) $\wedge$minimal(reliability(G)) $\leq$ reliability(s_{il}) $\leq$ optimal(reliability(G)) $\wedge$available(s_{ij}) $\wedge$ available(s_{il})
Trigger	reliability(Par_or(s_{ij}, s_{il})) $\geq$ reliability(s_{ij}) $\wedge$reliability(Par_or(s_{ij}, s_{il})) $\geq$ reliability(s_{il})
Effect	bound(s_{ij}) $\wedge$ bound(s_{il})

When a user needs a high-reliability service, while every available service has a certain level of risk, the system may execute more than one atomic service (e.g., s_{ij} and s_{il}) concurrently (with 1 out of n synchronization) to meet the demand. This is shown as action "Par_or(s_{ij}, s_{il})" in Table 1.4.

The concepts defined above form a basic framework for representing the planning strategy of composite services. During the service composition process, user goals are likely to change, the environment variables are often unpredictable, and the service repository is constantly updated, so the service composition plan has to adapt to these changes. Strategies can be formed to decide what adaptive actions to be carried out, based on the user's goals to be satisfied, the environment conditions to be monitored, and the service repository dynamically changing. We provide three categories of strategies according to the type of change: goal-driven adaptation strategy, environment-triggered adaptation strategy, and service-association strategy.

- *Goal-driven strategy:* Given a softgoal G, if Quality(s_1,...,s_r) does not satisfy the user's minimal (Quality(G)), we take actions according to the service repository and the environment variables to satisfy the user's new softgoal G, if possible. This corresponds to the scenario when user requirements change at runtime. Generally, a goal-driven strategy influences the selection of the structure of an abstract service.
- *Environment-triggered strategy:* Given a certain runtime context E, where there are composed but not yet executed services, if a certain constraint on E does not hold, alternative services are selected to satisfy the environment constraints. The strategy fits the cases where the environment is not stable and requires reconfiguration of concrete services.
- *Service-dominated strategy*: This strategy applies if there is a bounded set of services that haven't been executed yet and there is a new service providing equivalent function and better quality or one of the bounded services becomes unavailable. Assuming that user goals and the environment stay the same, the new service can satisfy user goals better, so we take action to add the new service or remove the unavailable one. This strategy is useful for a dynamically changing service repository and will result in replacement of the concrete services.

1.2.2.2 Adaptive Service Composition Process

This section introduces a generic adaptive service composition process (Fig. 1.2). It is closely related to the MAPE-K feedback loop, proposed by IBM in Autonomous Computing White Paper [16], in which "M" corresponds to P7 (monitor for events), "A" corresponds to P6 (analyze environmental variables), "P" refers to P4 (select and adapt predefined services, in other words, planning), "E" is P5 (execute selected services), and "K" includes P1, P2, where initial knowledge are acquired and domain information is interpreted and categorized. Next we introduce these proposed functions in detail.

Step 1: The initialization function (P1) receives inputs from the environment and the users, such as the system goals and the values of environmental variables.

Step 2: The domain information is then interpreted and categorized (P2). The required goals are used to build the goal model whose leaf goals can be matched with service functions, to select predefined services. The environmental variables are monitored according to the process discussed below.

Step 3: The service repository is categorized according to a set of abstract services which include many concrete atomic services available over the Internet. The Match abstract services operation (P3) matches the abstract service $\{S_0, \ldots, S_n, e_1, \ldots, e_n\}$ from the service repository with $\{F_0, \ldots, F_m, e_1, \ldots, e_n\}$ refined iteratively from the user's goals (*Function (G)*).

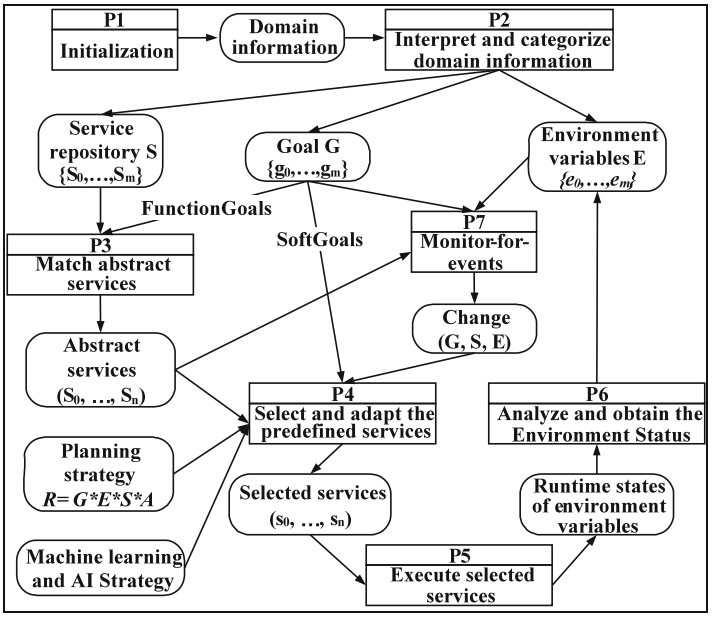

Fig. 1.2 A generic monitoring and adaptation process model

Step 4: The select and adapt concrete services operation (P4) selects atomic services or compositions from the abstract service according to user softgoals, Quality(G). If the Monitor operation (P7) identifies certain Change(S, G, E), the Select operation (P4) undertakes adaptive action to tackle the problems according to the current strategy ($G \times E \times S \times A$). During selection and adaptation, machine learning and AI strategies can be adopted to provide useful feedbacks. The select and adapt the concrete services (P4) would output a series of selected services.

Step 5: Selected services generated in Step 4 are now executed (P5). The execution affects the runtime values of the environmental variables, whose change can be monitored by the monitoring mechanism. And when abnormal behaviors have been observed, the system will trigger predefined actions to maintain the expected service effect.

Step 6: The runtime status is analyzed, and new environmental variables may be obtained (P4). If current values do not satisfy expected values of environmental variables, the monitor for events operation (P7) identifies the problem.

Step 7: During the execution of the service process (P5), user goals are also allowed to change, environmental variables are allowed to have uncertain changes, while service repository is also allowed to change dynamically. These changes (S, G, E) need to be captured and submitted to the select and adaptation engine for concrete services (P4).

As shown in Fig. 1.2, the steps P4, P5, P6, and P7 form a feedback loop that executes iteratively to support runtime adaptation. Those changes on the goals and the service repository come from the user and the platform correspondingly, so they are not part of the feedback loop.

In summary, the goal-driven runtime service composition adaptation proposed in this chapter aims to provide a different viewpoint and interpretation to runtime requirement monitoring and services adaptation. To allow runtime requirements changes and system adaptation, the software system needs to maintain a goal refinement model during execution, in which the set of user goals, environment variables, possible system actions, and triggering rules connecting the previous three sets can evolve dynamically. The proposed service adaptation architecture, the different types of rules in the rule base, the monitoring mechanism for collecting the environmental parameters, and the planning mechanism reasoning about adaptation strategies are to be further examined in our ongoing future work.

1.2.3 Human Factors

An engineered adaptive system captures the knowledge of human experts such as business analysts and architects as runtime models and adaptation rules and uses them as a knowledge base for adaptation decisions. On the other hand, most of

*Contributed by Xin Peng.

the software systems today can be treated as the so-called socio-technical systems, in which human, hardware, and software components work in tandem to fulfill stakeholder requirements [40]. Therefore, human factors are critical for the design and engineering of adaptive software systems.

Human factors in an adaptive system can be understood and considered from different perspectives. A human can act as an expert providing knowledge for runtime adaptation decisions. A human can also be treated as a user, an agent, or a component of an adaptive system. Human factors from these perspectives need to be considered together with software techniques such as reconfigurable software architectures when designing an adaptive system.

1.2.3.1 Human as Expert

In traditional software evolution, human experts such as business analysts and architects make adaptation decisions based on their business and design knowledge. Runtime adaptation can be regarded as the automation of human-directed adaptation based on runtime representations of human knowledge. Some adaptive systems even can have humans in the adaptation loop, where major changes are demanded and require human approval or guidance [41]. Moreover, runtime adaptation often involves both requirements and architectural decisions where different concerns require different knowledge [13]. Considering human as expert, we need to capture human knowledge at different layers as knowledge bases of runtime adaptation and establish some kinds of multilayered control and feedback loops at runtime. A difficulty of this multilayered loop lies in the complex interaction between different layers. For example, if a problem (e.g., performance degradation) cannot be handled by lower-layer adaptation, an adaptation request will be thrown to higher layers. On the other hand, higher-layer adaptation actions need to be mapped to lower-layer adaptation actions. This mapping involves complex traceability between different layers (e.g., requirements and architecture).

1.2.3.2 Human as User

Online systems such as online shopping and games usually serve a large number of users simultaneously. Each user in this kind of system has his own experience on the system. And their preferences on quality attributes are usually different. For example, a response time of 10 s is usually unacceptable for a young person doing online shopping but may be good enough for an old person. From this perspective, an adaptive system can plan the overall adaptation by considering personalized experiences of different users. This can provide more flexibility for system-level adaptation decisions. For example, the system can allocate less resources for users who are less sensitive to response time to ensure the satisfaction of more sensitive users. Considering human as user, an adaptive system can learn user experience and quality preference by implicitly monitoring user behaviors and feedback using HCI

(human-computer interaction) techniques such as eye tracking and touch sensors. Based on the learned user experience and preference, the system can make more flexible and personalized adaptations, for example, by using personalized utility functions for different users.

1.2.3.3 Human as Agent

An adaptive system can be an open system if it has multiple autonomous and heterogeneous participants (e.g., organizations and humans) interacting with each other in order to fulfill their respective goals [18]. Runtime adaptation in this kind of system involves not only technical solutions of individual participants but also social interactions among different participants. Considering human (also organization) as agent, we need to support the multi-agent nature of these systems to allow each agent to achieve its goals by both its own capabilities and social collaborations with others. To this end, the system needs to be designed to support decentralized requirements monitoring, reasoning, and adaptation using inter-agent interactions [40]. Each agent in the system can reason about the fulfillment and adaptation of its goals by considering inter-agent interactions such as label propagations and agent substitution. To decouple the specifications of different agents, commitments among participating agents need to be modeled and managed as a contract among them. By commitments, an agent has only to enter into the appropriate commitment relationships with another agent and need not care if the latter actually has the intention, since commitments are publicly verifiable and thus socially binding [18].

1.2.3.4 Human as Component

In a socio-technical system, humans are no longer just the users but an integral part of the system [21]. From this perspective, humans in a system can be treated as components that have their own capabilities and communicate and coordinate with other human/software/hardware components via various interfaces. This perspective is different from the perspective of human as agent: the former emphasizes the autonomous nature of humans and focuses on the goals of individual agents, while the latter emphasizes the achievement of the overall goals of a whole system with humans as components. Human as component can be reflected by the concept of human architecture [21], which describes the system's users in terms of human components and collaboration connectors along with their means of communication and coordination. Considering human as component, an adaptive system needs to integrate human components and their collaboration topology at all stages of the MAPE-K adaptation cycle [21].

1.3 Engineering Adaptive Software Systems

As with design, engineering adaptive software systems, that is realizing the design as an effective executing system on a specific platform, requires novel methods and tools focused on the adaptation mechanisms in the system. We require a way to represent important ingredients of adaptive software systems such as system structure, system behavior, user needs, system workload, or the surrounding environment, and modeling plays an important role in providing abstractions of the ingredients as first-class artifacts of system development, as well as mechanisms for manipulating them to cope with reuse and evolution. Provided with suitable connections with the system under study, Hidaka, Hu, and Martin show that the notion of model management from database research [7] can embody the adaptation process of software systems and support adaptive system development.

The final topic of this chapter focuses on the continuum of the adaptation process. With the advent of mature version control and backup systems, rollbacks of adaptations to past instances of a system are possible. Yu introduces the concept of future-proof requirements and uses them as a basis for an approach to derive system instances for future adaptations from previous instances.

1.3.1 Model Management in Adaptive Software Systems

Given a view of models as manipulable representations as first-class artifacts, we propose an approach to handling models using model management to embody the adaptation process of software systems.

We first introduce the model management concepts and how we can construct mappings between components in the MAPE-K loop. Update propagation in the MAPE-K loop is also shown to be achieved using model management operators.

Evolution of different components should be synchronized, and model management can cope with this synchronization. However, current model management does not have a clear guarantee of consistency because of its high degree of expressiveness. Within the synchronized adaptation, consistency could be maintained via the notion of bidirectional model transformations. We provide a prospective viewpoint toward extending current bidirectional transformation with the more complex scenario of changing mappings to cope with model management problems in self-adaptive systems.

1.3.1.1 Model Management

Self-adaptation mechanisms address different aspects of a system including performance, health, security, and configuration. In all these different kinds of mecha-

*(Contributed by Soichiro Hidaka, Zhenjiang Hu, and Patrick Martin).

nisms, precise models of aspects such as system structure, system behavior, user needs, system workload, or the surrounding environment are needed to perform the adaptation [14]. Models are therefore an important aspect of self-adaptive systems, and their development, testing, and evolution should be considered during the corresponding phases of the managed system's life cycle. Given this view of models as first-class artifacts of system development, it is natural to consider how to support features such as model reuse and evolution. We propose an approach to handling the models in self-adaptive systems based on the concept of model management [6, 7].

Model management, originally proposed by Bernstein et al. [7], is intended to help with managing change in models and with the transformation of data from one model to another. It treats models and mappings as first-class objects and defines high-level operations that simplify their use. We argue that model management can be adapted to the management of the models developed for self-adaptive systems and so facilitates a common understanding of the models and their concepts, interoperability and interaction among different models, and effective development of self-adaptive systems through the reuse and adaptation of existing models and mappings.

In the model management work, *models* are complex discreet structures that represent artifacts such as XML DTDs, website schemas, interface definitions, relational schemas, workflow definitions, and software configurations. At an abstract level, models can be viewed as directed graphs. The nodes in the graph are objects composed of properties, and the edges represent the containment relationship among the objects. The set of values of the properties describe the state of the object. Nodes are instances of a class, and each class definition describes the properties and relationships of instances of that class. More specifically, a model is a set of objects O that is identified by a root object r in O such that $O - \{r\}$ is the set of objects reachable from r by following the containment relationships. This simple representation of a model can be used to capture a large portion of models in self-adaptive systems, which are commonly represented in markup formats such as XML, XMI, JSON, and PMML.

Mappings are also first-class entities in model management. A mapping consists of connections between instances of two models. It defines transformations of the properties from the objects in the domain model to properties of the objects in the range model. A mapping is in fact represented as a model and so can be manipulated with similar operations.

The *operations* on models are intended to reduce the amount of programming required to manipulate models. Each operation should return a model so that operations can be composed. Each operation should also be generic so that it can be applied to any type of model. The basic operations defined by Bernstein et al. [7] include the following:

- **Match**: $\mathsf{Match}(M_1, M_2)$ takes two models M_1 and M_2 as input and returns a mapping between them. This could be defined as simply as returning the set of objects from M_1 and M_2 that match exactly or could use more complex notions of similarity.

- **Compose**: $map_1 \circ map_2$ takes two mappings map_1 and map_2 as input and returns their composition.
- **Merge**: $\mathsf{Merge}(M_1, M_2, map)$ takes two models M_1 and M_2 and a mapping *map* between them as input and returns a model that is the merge of M_1 and M_2 using *map* to guide the alignment of objects in the merged model.
- **Difference**: $\mathsf{Diff}(M_1, M_2)$ takes two models M_1 and M_2 as input and returns a model that contains the objects of M_1 that are not in M_2.
- **Copy**: $\mathsf{DeepCopy}(M, map)$ takes a model M and a mapping *map* as input (where the mapping is incident to the model) and returns a copy of both the model and mapping as output.
- **Enumerate**: $\mathsf{Enum}(M, next)$ takes a model M as input and performs a traversal of all the objects in M. The traversal order can be given as a parameter *next*, e.g., breadth-first or depth-first.
- **Model Generation**: $\mathsf{ModelGen}_{T_1,T_2}(M_1)$ takes a model M_1 conforming to the metamodel T_1 and returns a model conforming to the metamodel T_2 and a mapping between M_1 and M_2. Note that a metamodel specifies an abstract language that describes the model that conforms the metamodel. For example, if we consider a specific database state as a model, then its metamodel corresponds to the schema of the database state. If we consider a schema as a model, then the generic notion of relational data model that describe the schema corresponds to the metamodel of the schema.

Note that since models are object structures, they can be manipulated by the usual object-at-a-time operators: read an attribute, traverse a relationship, create an object, update an attribute, add or remove a relationship, etc.

Our use of model management in self-adaptive systems shares common goals with the work by Lehmann et al. on meta-modeling runtime models [34]. They propose a meta-metamodel to express metamodels of various runtime models. Using this common modeling approach facilitates achieving goals such as a common understanding of runtime models, a means for comparing different runtime models, and interoperability of runtime models. Our approach goes beyond these goals to support the development of self-adaptive systems through model reuse and adaptation.

1.3.1.2 Model Management for Implementing MAPE Loops

The MAPE-K (Monitor-Analyze-Plan-Execute) loop [32], as shown in Fig. 1.3 (left), is the heart of any self-adaptive system. The monitor part provides the mechanisms that collect, aggregate, filter, and report information collected from managed resources. The analyze part contains the mechanisms that correlate and model complex adaptation situations. The plan function encloses the mechanisms that construct the actions needed to achieve goals and objectives. The execute function groups the mechanisms that control the execution of an adaptation plan with considerations for dynamic updates. The knowledge is shared among the monitor, analyze, plan, and execute functions.

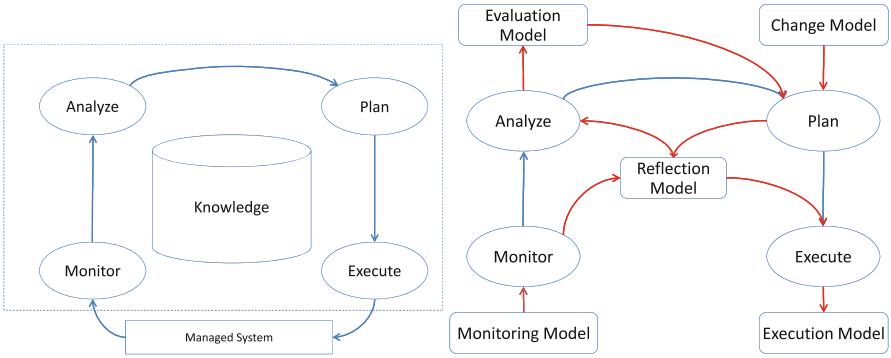

Fig. 1.3 MAPE-K loop (left), runtime models (right)

It is shown in [48] that knowledge in the MAPE-K loop can be refined to a set of runtime models, as shown in Fig. 1.3 (right). *Monitoring Models* map system-level observations to the abstraction level of *Reflection Models*. The Reflection Models are analyzed to identify adaptation needs by applying *Evaluation Models* that, e.g., define constraints on Reflection Models. If adaptation needs have been identified, the planning activity devises a plan prescribing the adaptation on the Reflection Models. Planning is specified by *Change Models* describing the adaptable software's variability space. Evaluation Models such as utility preferences guide the exploration of this space to find an appropriate adaptation. Finally, the execute activity enacts the planned adaptation on the adaptable software based on *Execution Models* that refine model-level adaptation to system-level adaptation.

Model management enables us to easily establish static mappings among the runtime models in Fig. 1.3 (right). For the static mappings of Monitor, Analyze, and Execute, the definitions are straightforward.

$$
\begin{aligned}
map_{\text{Monitor}} &= \mathsf{Match}(\text{Monitoring Model, Reflection Model}) \\
map_{\text{Analyze}} &= \mathsf{Match}(\text{Reflection Model, Evaluation Model}) \\
map_{\text{Execute}} &= \mathsf{Match}(\text{Reflection Model, Execution Model})
\end{aligned}
$$

For the mapping of Plan, we first merge Evaluation Model and Change Model to form a new model, say Adaptation Model, by

$$\text{Adaptation Model} = \mathsf{Merge}(\text{Evaluation Model, Change Model}, map_{EC})$$

where $map_{EC} = \mathsf{Match}(\text{Evaluation Model, Change Model})$, and then we build a mapping between Adaptation Model and Reflection Model using Match.

Furthermore, model management provides a convenient way to implement dynamic update propagation along the MAPE-K loop. As an example, consider that the Monitoring Model is updated to M' and that we want to propagate this update to the Reflection Model and get R' together with a corresponding mapping between

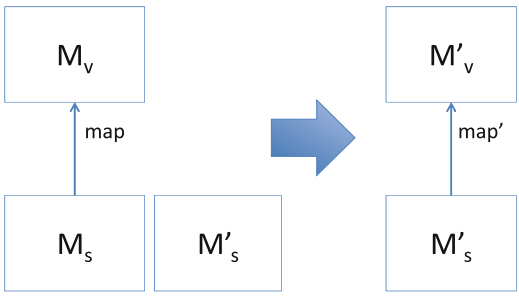

Fig. 1.4 Update propagation from M_s to M_v

M' and R'. To see how to implement this update propagation, let us consider a more general case. Suppose that we are given two models M_v and M_s and a mapping *map* that maps elements of M_s to that of M_v (see Fig. 1.4). Given that M_s is updated to M'_s, the problem is to define an updated version M'_v of M_v that is consistent with M'_s with a new mapping map' from M'_s to M'_v. We can solve this problem using model management operators as follows:

$map_3 = \mathsf{Match}(M_s, M'_s)$
$map_4 = map \circ map_3$
$(M'_v, map') = \mathsf{DeepCopy}(M_v, map_4)$
$(M''_v, map_5) = \mathsf{Diff}(M'_v, map')$
For each e in $\mathsf{Enum}(map_5, \text{depthFirst})$, delete the source element from M'_v.

We omit the detailed explanation of the above program. In fact, it is very similar to the solution to the schema evolution problem in [7].

1.3.1.3 Bidirectional Model Transformations for Adaptive Software Systems

While model management facilitates the propagation of changes across models, in order to make self-adaptive systems stable, the models together with mappings between them should satisfy some properties. For example, in the scenario of update propagation depicted in Fig. 1.4, after propagation is completed, M'_s and M'_v should be related by map' in a way that is consistent with the way that *map* related M_s and M_v. Moreover, if the M_v that is the target of transformation via *map* is updated to M'_v, the update should to be propagated back to M_s so that updated model M'_s still generates M'_v via transformation.

To guarantee these *round-trip* properties, bidirectional transformations [17] between models can be used. Song et al. [45] identify some of these transformations in the context of synchronization between an architecture model and the system under study for adaptation to environmental changes. Once a bidirectional transformation is defined between models, changes on these models are propagated

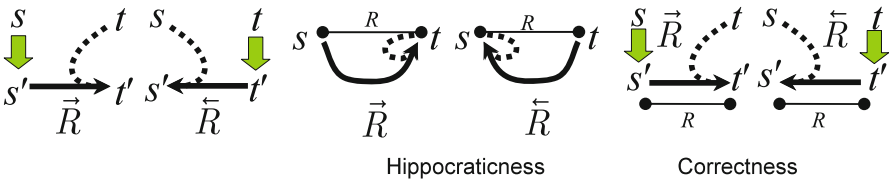

Fig. 1.5 Symmetric bidirectional transformation (maintainer) scheme and some major properties

back and forth between the models. If the transformation creates a view model from another model, the model can be managed through the views. To achieve consistency, bidirectional properties (often called well-behavedness) are studied to guarantee that updates are correctly preserved in the process of transformations. Traces that are computed along with the transformation help propagate these changes.

Bidirectional transformation respects the user-defined relations between those parts (system model and architecture model in [45]) and propagates changes over the relations.

Figure 1.5 shows the symmetric bidirectional transformation (maintainer) scheme. $x \bullet\!\!-\!\!\bullet\, y$ indicates x and y are consistent.

$\overrightarrow{R} : S \times T \to T$ for the relation R produces from old target t and updated source s' the new target t', i.e., $\overrightarrow{R}(s', t) = t'$. $\overleftarrow{R} : S \times T \to S$ is defined symmetrically. Hippocraticness [46] says that transformation from already consistent pair lead to no modification on the counterpart artifacts. For the forward direction, $(s, t) \in R \Rightarrow \overrightarrow{R}(s, t) = t$. Correctness [46] says that after updates are propagated by transformations in both directions, the resultant pairs are consistent. For the forward direction, $\overrightarrow{R}(s', t) \in R$.

Bernstein [4] demonstrates that model management supports round-trip engineering (e.g., model-code coevolution), by the scenario of updating the M_v side in Fig. 1.4 as mentioned above. However, it is not explicit in [4] on the formal guarantee of the round-trip engineering, while bidirectional transformations guarantee the well-behavedness property, keeping mappings constant. The main reason for this gap is that the model management framework is more expressive in the sense that mappings are not fixed but manipulable and sometimes automatically generated. On the contrary, bidirectional transformation currently tackles problems where mappings are fixed, like view update problems where queries (transformations) to generate views are fixed. View update problems are considered as a special case of mapping generations in [5] among more general model management scenarios including changing the transformations as well as models.

We discuss here how different kinds of updates could be accommodated by model management, using Fig. 1.4.

For in-place updates of model elements, it does not cause metamodel changes, so map_3 generated by the first step using Match() may produce an "almost-identity" mapping except that, in the element of map_3 that is responsible for relating the

element before and after updates, an equivalence operator that defines these element to be equal is generated. Therefore, composition in the second step is almost trivial. Backward transformation propagating changes from M_v goes through this operator.

For deletion of model elements, model management changes the map by deleting the corresponding elements in the maps. From the viewpoint of bidirectional transformations, it corresponds to dropping a bidirectional transformation rule that involved the transformation of the deleted elements (you can consider dropping ATL rules that is translated to bidirectional graph transformation in the framework of Sasano et al. [44]).

In the case of inserting new model elements, the operator $\mathsf{ModelGen}_{MM_v,MM_s}()$ where MM_v and MM_s, respectively, denote the metamodels to which the models M_v and M_s conform is utilized as the default mapping to support backward transformation for a newly inserted object on the M_v side, reverse engineering the corresponding element in M_s, and that is the invariant that corresponds to the mapping in bidirectional transformation. As a postcondition, $M'_v = \mathsf{ModelGen}_{MM_s,MM_v}(M'_s)$ should be satisfied. In the context of bidirectional transformations, this kind of operation is either explicitly provided (create function in [25]) or derived (insertion handling in [28]) solely relying on a given transformation using generic inversion strategy like Universal Resolving Algorithm [1].

Toward bidirectional model transformation capable of wider range of changing scenarios, bidirectional transformations should also consider scenarios in which transformation evolves. For example, metamodel evolution scenario forces transformations to evolve. A recent study by Hoisl et al. [29] considers coevolution of metamodels to cope with transformation changes, using the notion of higher-order transformation [47]. However, bidirectional transformation property under this scenario is not explicitly addressed so far, and we could easily imagine part of the reason being too many degrees of freedom introduced by allowing changes of the mapping. The strategies like those mentioned in Sect. 1.2.2 might guide reasoning about the property.

1.3.2 Adaptation for Evolving Software Systems

Time travel is the dream of archaeologists to verify their hypotheses of our history and the subject of science fiction writers in order to speculate on our destiny. Supported by change management and backup file systems, however, imperfect, traveling to the past of a project is no longer a dream. However, traveling to the future software world remains largely so. On the basis of various advances in the theory and practice of requirements engineering [43, 49, 56], especially the progress in the active maintenance of runtime traceability relationships [55, 57] between problems and solutions, we speculate that traveling to the future may be

*(Contributed by Yijun Yu).

enabled by (a collection of) carefully designed software systems [51]. We present a scenario where some of these requirements can be made future-proof through a well-designed adaptive systems. We also present a few challenges faced by such a design for some other requirements to be future-proof.

1.3.2.1 From Natural Selection to Self-Adaptation

Through natural selection, biological species survive environmental changes by passing on the fittest features over generations [19]. To survive the test of time, software systems also have to respond to a changing environment to better satisfy stakeholders' core requirements [43]. Bacteria of the simplest life form coexist with advanced mammals as long as suitable environments for them still exist after billions years of profound changes. The ecosystem of the biological world has its match in the software world too: early generations of command-line programs coexist with the bells and whistles of mobile apps, and they probably fit in the present high-assurance-requiring purposes like steering pathfinders to the deep space better.

Software archaeologists [8] attempt to find the fossils of historical software such that knowledge of legacy could be preserved and lessons could be learnt for developing modern software [50]. The motivation for reverse engineering software requirements from legacy code [56] is to achieve better understanding of the traceability [27] between the current artifacts to those of the past.

If all that is possible, what about the *time travel* then? Compared to teleporting objects obeying physics laws [22], it is much easier for software code to travel to the past, as long as one can reconstruct the contemporary version of the operating environment. One can recover almost every artifact committed to software change management (SCM) systems. With the advent of backup file systems such as the trademarked *Time Machine* of MacOSX, not only code but also related files can be retrieved. Ignoring the fact that much of the design rationale and documentation can be missing from the SCM or backup files, for the sake of argument, one can in principle teleport the current software system to its ancestors' environments. The basic requirement for such a time machine is to answer the following question:

When it comes to teleporting a software system to the future, is there a way to allow the current software system to run without worrying about possible environmental changes?

1.3.2.2 Future-Proof by Core Architectures

Our study here focuses on the open-source software systems such as Eclipse that have survived years of rapid changes, and the analysis is reproducible. To establish the ground truths about future-proof software systems, we have taken a close look at the Eclipse SDK and its ecosystem in a prior study [51]. As a whole the system underwent changes all the time. However, the architectural design of the software system is quite stable. Known as the *plug-in architecture*, through the introduction of

structural extensions, API compatibility has been extended to support compatibility between data schema and control classes. On the other hand, one cannot see such stability by looking inside a plug-in component: the number of Java classes grows at a superlinear speed, while more methods are introduced release after release. One insight we obtained from the empirical study was the persistence of a stable *core* subset of plug-in components (i.e., 23 of them) since the very first release 1.0; the interface dependencies among these core components are preserved along all releases.

We first define the concept of *future-proof software requirement* and test on the special components of the core architecture of Eclipse SDK. Then we propose to redesign the plug-in architecture as an adaptive time-managed system such that components of the older releases can work along with the newer ones after introducing time-managed wrappers.

1.3.2.3 Future-Proof Requirements

Consider the basic requirement problem characterized by Zave and Jackson [58]:

$$W, S \vdash R \tag{1.1}$$

where descriptions W indicate the properties of the domains in the physical world (i.e., environment), in which a solution S brings satisfaction to the requirement R.

Naturally, a temporal extension of the problem is

$$W(t), S(t) \vdash R(t) \tag{1.2}$$

where t stands for a given time.

From the (current) time t_0 onward to the time t in the future ($t > t_0$), a *future-proof solution* is defined as the solution $S(t_0)$ that brings satisfaction to the future requirement:

$$W(t), S(t_0) \vdash R(t) \tag{1.3}$$

If the current solution $S(t_0)$ does not satisfy the requirement of the future, i.e., $W(t), S(t_0) \nvdash R(t)$, what shall be done then?

1.3.2.4 Composition Requirements for Encapsulating Future Changes

Here we argue that there is a wrapper *circa* with which the composed solution circa $S(t_0)$ satisfies:

$$W(t), \text{ circa } S(t_0) \vdash R(t) \tag{1.4}$$

Wrapping mechanism at the design (solution) level such as feature configurations, state-chart behavior models, or Koala component-connector models offers

the capability to switch from one alternative to another [54]. To make such high-variability design work, a monitoring and switching (or self-reconfiguring) mechanism is required for the system to satisfy the same requirements [42]. However, at runtime or after evolution, the original requirements can be relaxed [52], evolved [23], or adapted [2]. Therefore, we state that the future-proof requirement for solving a problem is to be able to adapt the solution to the changes of both the requirement and the environment. The scope of the environment depends largely on where the system boundary is drawn. A complex software system consists of multiple components; each addresses a subproblem of the whole. When composed together, one ought to consider at least two basic requirement problems, with one solution being a designed domain in the environment of the other.

Without loss of generality, let us consider two such components S_1 and S_2 and assuming that S_2 is a designed domain (i.e., prerequisite) for S_1:

$$\begin{cases} t : W_1, S_1 \vdash R_1 \\ \quad W_2, S_2 \vdash R_2 \\ \quad W_1 \vdash W_2 \wedge S_2 \end{cases} \tag{1.5}$$

This creates a dynamic dependency between $S_2(t)$ and $S_1(t)$. Unfortunately, as S_1 and S_2 are usually managed separately by different teams, they have their own pace and criteria of evolution. Therefore, for the designer of the composed system, the existence of such dependencies generates an obligation to maintain their consistency.

If components in the complex ecosystem and their dependencies can be versioned, Ma et al. [38] propose transactional constraints to maintain version consistency on dependencies between the coevolving components such that components only execute with compatible ones, regardless of their individual update transactions. The transactional constraints approximate the dynamic dependencies by introducing *future* and *past* dependency edges to compute a consistent configuration. The *future validity* is the satisfaction of the dependency after a transaction starts, and the *past validity* is the satisfaction of the dependency after a transaction finishes.

To have the entire system travel to the future or to make a component of it future-proof, all components need to be cooperative, i.e., following the same transaction model to update their consistency requirements locally. Ma et al. [38] further simulate their proposed version consistency update model in comparison to the quiescence update condition [33]. It is yet to be decided where to obtain the future/past dependency edges and how to fix them if the classification is imprecise.

1.3.2.5 Evolving Component-Based Systems: Addressing the Meaningful Changes

Consider a large-scale evolving component-based system where profound changes can happen to any part of the system, e.g., to each component at the level of

requirements, architecture, design, code, and test cases. The GMF project of the Eclipse ecosystem itself has 17 K changes committed to the repository [57], not to mention the large number of changes outside the repository and the large number of projects related to it. Due to these frequent changes, the dependencies between components and the answer to the requirement validation and future-proof requirement problems may change as well.

With n components connected to at least one other component(s), the system dependency graph would have a complexity of $O(n)$ to $O(n^2)$ edges. A typical release of Eclipse SDK has 500 plug-in components, and a typical distribution of the Linux has 30 K software packages. Therefore, instead of checking the versioned consistency for all pairs of changes, we propose to apply a filter for meaningful change to detect whether an update is harmful for adaptation.

Generalizing from earlier work on meaningful change detection [55], a change to the requirement problem is considered meaningful if and only if the following two relations hold:

$$\begin{cases} t_1 : W, S \vdash R \\ t_2 : W, S \nvdash R \end{cases} \tag{1.6}$$

where $t_1 \neq t_2$ are two time stamps at either side of the change. Otherwise, the change does not require adaptation of the solution.

When it comes to two dependent components, a change that is not meaningful for one might be meaningful to the other.

For example, even if a change is not meaningful to the component S_2, it may cause a denial of (1.5) if

$$t_2 : W_1 \nvdash W_2 \wedge S_2 \tag{1.7}$$

To check the violation of future-proof requirements, one can ignore all the changes that do not satisfy (1.6) for individual components or (1.7) for dynamic component dependencies.

1.3.2.6 Example: Meeting Scheduler

To illustrate the concepts defined above, let us consider a simplified example: Busy Hat is a software development company, whose secretary's job of scheduling meetings is replaced by a computer program. The program has two components initially. One collects time constraints from individual employees (TimeCollector); the other sends reminders to notify the employees who wanted to participate in a given meeting (Reminder). Both components make use of a subcomponent Email to notify the people involved.

The TimeCollector also uses Spreadsheet to fill in the information. The (much simplified) requirement problems are stated as Fig. 1.6.

$$
\begin{cases}
t_1 : \begin{cases} Participant, Meeting, Email, Reminder & \vdash ParticipantsNotified \\ Recepient, Info, Email & \vdash InfoSent \\ Participant, Meeting, NotificationEmailSent & \vdash ParticipantsNotified \end{cases} & (v_1^{(1)}) \\
t_2 : \begin{cases} Employee, Email, Spreadsheet, TimeCollector & \vdash ConstraintsCollected \\ Employee, ElicitationEmailSent, SheetFilled & \vdash ConstraintsCollected \end{cases} & (v_1^{(2)})
\end{cases}
$$

Fig. 1.6 Initial requirement problems

$$
\begin{cases}
t_3 : \begin{cases} Employee, Web, Form, TimeCollector & \vdash ConstraintsCollected \\ Employee, ElicitationEmailSent, FormFilled & \vdash ConstraintsCollected \end{cases} & (v_2^{(1)}) \\
t_4 : \begin{cases} Participant, Meeting, SMS, Reminder & \vdash ParticipantsNotified \\ Recepient, Info, Web, Form & \vdash InfoSent \\ Employee, ElicitationInfoSent, FormFilled & \vdash ConstraintsCollected \\ Participant, Meeting, NotificationSent & \vdash ParticipantsNotified \end{cases} & (v_2^{(2)}) \\
t_5 : \begin{cases} Employee, Web, iCal, TimeCollector & \vdash ConstraintsCollected \\ Employee, ElicitationInfoSent, iCalFetched & \vdash ConstraintsCollected \end{cases} & (v_3^{(1)}) \\
t_6 : \begin{cases} Participant, Meeting, Twitter, Reminder & \vdash ParticipantsNotified \wedge Size < 140 \\ Recepient, Info, Twitter & \vdash InfoSent \end{cases} & (v_3^{(2)})
\end{cases}
$$

Fig. 1.7 Adaptation to evolved requirements and environments

Different labels for the v_1s here emphasize their different time stamps. The interface between an employee and the time collector is the method *Constraint[] TimeCollector.fillSheet(EmployID id)* in the API that takes the employee ID as input, via interaction with the Employee, and outputs the constraints corresponding to the rows in the time table. Similarly, the initial interface between the reminder and the participants of a meeting is the API method *Reminder.sendEmail(Participant p, Meeting m)*.

Consider the situation where two changes arise in each of the two components (see Fig. 1.7), which are developed in tandem. At $v_2^{(1)}$ of TimeCollector, it uses Web forms instead to collect the individual constraints, and at $v_3^{(1)}$ of TimeCollector, it supports importing iCal entries, such as Google Calendars, into the schema. At an independent pace, $v_2^{(2)}$ of Reminder uses a SMS service to send a reminder rather than using emails, and at $v_3^{(2)}$ it further limits the length of short messages to 140 characters to use Twitter instead.

Let us denote the time stamps for the six versions as t_1 to t_6, respectively, assuming that the six versions are interleaving, say, $t_1 < t_2 < t_3 < t_4 < t_5 < t_6$.

To illustrate the concept of dynamic dependencies, we assume that the Reminder component is also responsible for reminding employees to collect time tables for the interaction. The reused Email component is designed for the TimeCollector component. In other words, there is a dependency from TimeCollector to Email.

Checking the interfaces of the two components, one realizes that at t_2 the dynamic dependency is reflected by sharing email client as the designed domain. However, at t_3 a new interface between the Web form used by TimeCollector and the

Email client used by Reminder needs to be configured. At t_4, the interface between the Web form and the SMS service replaces that pair. At t_5 and t_6, the roles of Web form and SMS are substituted by iCal and Twitter.

Although the dynamic dependencies in this example persist over all revisions, they shall be carefully maintained. Otherwise, the design is not future-proof, that is, besides possible invalidation of functional requirements local to the components, the changes of interfaces may also invalidate the dynamic dependency required between the two components.

Further to these precautions, it is clear, at least at t_6, that the functional requirement of Reminder is updated, e.g., to include the 140 character restriction. The restriction is a change that was not predicted earlier. If the solution had not changed from Email to Twitter, such a restriction would not have made sense earlier. Therefore, the changes to validation (e.g., test cases) shall be included as part of the future-proof (i.e., forward compatibility in this case) contract.

To keep the example simple, all the changes are meaningful to the future-proof requirements. Of course, you can imagine hundreds of detailed changes in design, implementation, or even marketing, such as selecting which Email client(s) to support, deciding which carrier(s) of Telecom company to use, etc. Although these are relevant to quality requirements when explicitly modeled, for the simplicity and illustration purpose, here we regard them all as not meaningful to the functional requirement of meeting scheduling and the corresponding future-proof requirements.

1.3.2.7 Back to the Future

So far, we have presented the concept of future-proof software requirements which are derived from Zave and Jackson's basic requirements problem and manifested as the version consistency problems for updating component-based software systems. Having observed the mechanisms of representative component-based ecosystems, where the update dependencies have been versioned over the years, we show that it is possible to design an Adaptive Time Managed System through a prototype that augments the developer-oriented update mechanism.

1.4 Conclusion

Providing support in all phases of the life cycle of adaptive software systems is an important challenge facing the software engineering research community. This chapter highlighted current research on methods and techniques for the design and engineering of adaptive software systems.

We started by presenting decisions to be considered when designing predictive feedback loops of self-adaptive systems in the context of Software Defined Infrastructure typified by clouds. They were classified into the following three types of

deployments. *Observation* cluster concern includes what to monitor, the source and the format of the data, timing and frequency of monitoring in the Monitoring service, as well as what and how states are determined based on the monitored data and type of filters in the Filter service. *Representation* cluster concerns the runtime representation of quantitative dependencies among adaptation targets, perturbations, Actuator variables, measured metrics, and internal states in the Prediction Models. When and how to switch models that capture different kinds of QoS considerations are also concerned. *Control* cluster is concerned with the rich space of possible adaptation solutions and the mechanisms in the Controllers service, where Model predictive optimization connects the current state with desired states and goals, estimating the effect of different decisions using Prediction Models.

Next we showed that adaptation can be reified by goal-driven runtime service composition adaptation. Given a set of primitive services, the subset of these services are composed and reorganized by different adaptation strategy. When a particular service can meet better QoS than already deployed service, the Revoke action replaces the service. When combination of multiple services collectively achieve better throughput, these services are conjunctively (and) composed. When combination of multiple services collectively achieve better reliability, they are disjunctively (or) composed. They are issued based on three kinds of strategies, namely, *goal-driven*, *environment-triggered* and *service-demanded* strategies. Concrete steps are identified to implement MAPE-K loops.

Concerning human factors, four viewpoints are identified. *Human as experts* provides knowledge for making decision of runtime adaptation. Experts may participate in the adaptation loop to approve and guide the adaptation. *Human as user* are concerned with the consumers subject to observation. Systems can be adjusted to fulfill the utility of these users experiences and preferences by machine learning at the user interface like eye tracking. *Human as agent* are entities aiming at own objectives based on their own abilities and collaboration with other agents. Degree of objective achievements and adaptation of each agent can be reasoned about by label propagation and agent substitution. *Human as component* captures human as an integral part of the system, having their own ability to interact and cooperate with other human, software, and hardware components via various interfaces. They are distinguished from agents in that agents have their own objectives, while human as component aims at the overall goal of the system. Various levels of the MAPE-K adaptation cycle takes the human component and its collaboration topology into account.

Given the view of models as first-class artifacts representing elements in self-adaptive systems, we proposed an approach to handling models in self-adaptive systems. We argued that model management can be adapted to the management of the models developed for self-adaptive systems and effective development of self-adaptive systems can be achieved through reuse and adaptation of existing models and mappings. We have shown that model management can implement MAPE-K loops, and for their synchronized adaptation, consistency could be maintained via bidirectional transformation with future enhancement to cope with modifications of mappings.

Finally we have shown that sustainable adaptation process can be achieved by Adaptive Time Management System, identifying core architectures that can relatively stay stable and making requirements future-proof by introducing time axis to the predicates in Zave and Jackson's basic requirements problem, identifying meaningful changes thereof, illustrated by Meeting scheduler consisting of dynamically evolving interdependent components.

With the new challenges of ultra-large-scale and continuously available software, adaptive software systems are more important than ever. We believe that the methods and techniques highlighted in this chapter further stimulate the research on design and engineering of adaptive software systems, to push the limits in the adaptivity of software systems.

Acknowledgements We thank all the participants of NII Shonan Meeting Seminar No. 027 on Engineering Adaptive Software Systems (EASSy) for their valuable discussions with us to deliver this chapter. The authors of Sect. 1.2.2 receive partial financial support from the National Natural Science Foundation of China (No. 61033006).

References

1. Abramov, S.M., Glück, R.: Principles of inverse computation and the universal resolving algorithm. In: Mogensen, T.Æ. (eds.) The Essence of Computation, pp. 269–295. Springer, Berlin (2002)
2. Baresi, L., Pasquale, L.: Adaptation goals for adaptive service-oriented architectures. In: Avgeriou, P., Grundy, J., Hall, J.G., Lago, P., Mistrík, I. (eds.) Relating Software Requirements and Architectures, pp. 161–181. Springer, Berlin/Heidelberg. http://link.springer.com/chapter/10.1007/978-3-642-21001-3_10
3. Bencomo, N., Whittle, J., Sawyer, P., Finkelstein, A., Letier, E.: Requirements reflection: requirements as runtime entities. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, ICSE '10, vol. 2, pp. 199–202. ACM, New York (2010). https://doi.org/10.1145/1810295.1810329
4. Bernstein, P.A.: Applying model management to classical meta data problems. In: First Biennial Conference on Innovative Data Systems Research (CIDR 2003), Asilomar, CA, USA, 5–8 Jan (2003)
5. Bernstein, P.A., Melnik, S.: Model management 2.0: manipulating richer mappings. In: Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data, SIGMOD '07, Beijing, pp. 1–12. ACM, New York (2007). https://doi.org/10.1145/1247480.1247482
6. Bernstein, P.A., Rahm, E.: Data warehouse scenarios for model management. In: Proceedings of the 19th International Conference on Conceptual Modeling, ER'00, Salt Lake City, pp. 1–15. Springer, Berlin/Heidelberg (2000). http://dl.acm.org/citation.cfm?id=1765112.1765114
7. Bernstein, P.A., Halevy, A.Y., Pottinger, R.A.: A vision for management of complex models. SIGMOD Rec. **29**(4), 55–63 (2000). https://doi.org/10.1145/369275.369289
8. Booch, G.: Software archeology and the handbook of software architecture. In: Workshop on Software Reengineering, Bad Honnef, pp. 5–6 (2008). http://dblp.uni-trier.de/rec/bibtex/conf/wsr/Booch08
9. Brun, Y., et al.: Engineering self-adaptive systems through feedback loops. Lect. Notes Comput. Sci. **5525**, 48–70 (2009)

10. Brun, Y., Desmarais, R., Geihs, K., Litoiu, M., Lopes, A., Shaw, M., Smit, M.: A design space for self-adaptive systems. In: Lemos, R., Giese, H., Müller, Hausi, A., Shaw, M. (eds.) Software Engineering for Self-Adaptive Systems II. Lecture Notes in Computer Science, vol. 7475, pp. 33–50. Springer, Berlin/Heidelberg (2013). https://doi.org/10.1007/978-3-642-35813-5_2
11. Castro, J., Kolp, M., Mylopoulos, J.: Towards requirements-driven information systems engineering: the tropos project. Inf. Syst. **27**(6), 365–389 (2002). https://doi.org/10.1016/S0306-4379(02)00012-1
12. Chen, I.R., Bastani, F.B., Tsao, T.W.: On the reliability of ai planning software in real-time applications. IEEE Trans. Knowl. Data Eng. **7**(1), 4–13 (1995). https://doi.org/10.1109/69.368522
13. Chen, B., Peng, X., Yu, Y., Nuseibeh, B., Zhao, W.: Self-adaptation through incremental generative model transformations at runtime. In: 36th International Conference on Software Engineering (ICSE 2014), pp. 676–687. Hyderabad, India. ACM/IEEE (2014)
14. Cheng, B.H., Lemos, R., Giese, H., Inverardi, P., Magee, J., Andersson, J., Becker, B., Bencomo, N., Brun, Y., Cukic, B., Marzo Serugendo, G., Dustdar, S., Finkelstein, A., Gacek, C., Geihs, K., Grassi, V., Karsai, G., Kienle, H.M., Kramer, J., Litoiu, M., Malek, S., Mirandola, R., Müller, H.A., Park, S., Shaw, M., Tichy, M., Tivoli, M., Weyns, D., Whittle, J.: Software Engineering for Self-Adaptive Systems, chap. Software Engineering for Self-Adaptive Systems: A Research Roadmap, pp. 1–26. Springer, Berlin/Heidelberg (2009). https://doi.org/10.1007/978-3-642-02161-9_1
15. Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J. (eds.): Software Engineering for Self-Adaptive Systems. Lecture Notes in Computer Science, vol. 5525. Springer, Berlin/New York (2009)
16. IBM Corporation: An architectural blueprint for autonomic computing. Autonomic Computing White Paper, 4th edn. Technical report, IBM (2006)
17. Czarnecki, K., Foster, J.N., Hu, Z., Lämmel, R., Schürr, A., Terwilliger, J.F.: Bidirectional transformations: a cross-discipline perspective. In: ICMT'09, ETH Zurich, pp. 260–283 (2009)
18. Dalpiaz, F., Chopra, A.K., Giorgini, P., Mylopoulos, J.: Adaptation in open systems: giving interaction its rightful place. In: Proceedings of the 29th International Conference on Conceptual Modeling, pp. 31–45. ER'10, Vancouver. Springer, Berlin/Heidelberg (2010). http://dl.acm.org/citation.cfm?id=1929757.1929761
19. Darwin, C.: On the Origin of Species by Means of Natural Selection. D. Appleton and Co., New York (1859)
20. de Lemos, R., Giese, H., Müller, H.A., Shaw, M. (eds.): Software Engineering for Self-Adaptive Systems II – International Seminar, Dagstuhl Castle, 24–29 Oct 2010 Revised Selected and Invited Papers. Lecture Notes in Computer Science, vol. 7475. Springer (2013)
21. Dorn, C., Taylor, R.N.: Coupling software architecture and human architecture for collaboration-aware system adaptation. In: Proceedings of the 2013 International Conference on Software Engineering, ICSE '13, San Francisco, pp. 53–62. IEEE Press, Piscataway (2013). http://dl.acm.org/citation.cfm?id=2486788.2486796
22. Earman, J., Smeenk, C., Wüthrich, C.: Do the laws of physics forbid the operation of time machines? Synthese **169**(1), 91–124 (2009). http://link.springer.com/article/10.1007/s11229-008-9338-2
23. Ernst, N., Borgida, A., Jureta, I.: Finding incremental solutions for evolving requirements. In: Requirements Engineering Conference (RE), Trento, 2011 19th IEEE International, pp. 15–24 (2011)
24. Fickas, S., Feather, M.S.: Requirements monitoring in dynamic environments. In: Proceedings of the Second IEEE International Symposium on Requirements Engineering, RE '95, York, UK, p. 140. IEEE Computer Society, Washington, DC (1995). http://dl.acm.org/citation.cfm?id=827254.827800
25. Foster, J.N., Greenwald, M.B., Moore, J.T., Pierce, B.C., Schmitt, A.: Combinators for bidirectional tree transformations: a linguistic approach to the view-update problem. ACM Trans. Program. Lang. Syst. **29** (2007). https://doi.org/10.1145/1232420.1232424

26. Ghanbari, H., Litoiu, M.: Replica placement in cloud through simple stochastic model predictive control. In: IEEE Cloud, Anchorage, Alaska, 27 June–2 July (2014)
27. Gotel, O.C., Finkelstein, C.W.: An analysis of the requirements traceability problem. In: Proceedings of the First International Conference on Requirements Engineering, Colorado Springs, pp. 94–101. IEEE (1994)
28. Hidaka, S., Hu, Z., Inaba, K., Kato, H., Matsuda, K., Nakano, K.: Bidirectionalizing graph transformations. In: ACM SIGPLAN International Conference on Functional Programming, Baltimore, Maryland pp. 205–216. ACM (2010)
29. Hoisl, B., Hu, Z., Hidaka, S.: Towards co-evolution in model-driven development via bidirectional higher-order transformation. In: Proceedings of the 2nd International Conference on Model-Driven Engineering and Software Development, Lisbon, Portugal, pp. 466–471. SciTePress (2014). http://nm.wu-wien.ac.at/nm/file/MODELSWARD2014-PP%2epdf?m=download
30. Jian, Y., Li, T., Liu, L., Yu, E.: Goal-oriented requirements modelling for running systems. In: 2010 First International Workshop on Requirements@Run.Time (RE@RunTime), Sydney, NSW, Australia, pp. 1–8 (2010)
31. Jureta, I.J., Faulkner, S., Thiran, P.: Dynamic requirements specification for adaptable and open service-oriented systems. In: Proceedings of the 5th International Conference on Service-Oriented Computing, ICSOC '07, Vienna, Austria, pp. 270–282. Springer, Berlin/Heidelberg (2007). https://doi.org/10.1007/978-3-540-74974-5_22
32. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. Computer **36**(1), 41–50 (2003). https://doi.org/10.1109/MC.2003.1160055
33. Kramer, J., Magee, J.: The evolving philosophers problem: dynamic change management. IEEE Trans. Softw. Eng. **16**(11), 1293–1306 (1990)
34. Lehmann, G., Blumendorf, M., Trollmann, F., Albayrak, S.: Meta-modeling runtime models. In: Proceedings of the 2010 International Conference on Models in Software Engineering, Oslo, Norway, pp. 209–223. MODELS'10. Springer, Berlin/Heidelberg (2011). http://dl.acm.org/citation.cfm?id=2008503.2008532
35. Li, J., Woodside, C.M., Chinneck, J., Litoiu, M.: Adaptive Cloud Deployment Using Persistence Strategies and Application Awareness, IEEE Trans. Cloud Comput. 5(2), pp. 277–290 (2017)
36. Li, J.Z., Woodside, M., Chinneck, J., Litoiu, M.: Cloudopt: multi-goal optimization of application deployments across a cloud. In: Proceedings of the 7th International Conference on Network and Services Management, Paris, France, pp. 162–170. CNSM '11. International Federation for Information Processing, Laxenburg (2011). http://dl.acm.org/citation.cfm?id=2147671.2147697
37. Liu, L., Liu, Q., Chi, C., Jin, Z., Yu, E.: Towards a service requirements modelling ontology based on agent knowledge and intentions. Int. J. Agent-Oriented Softw. Eng. **2**(3), 324–349 (2008). https://doi.org/10.1504/IJAOSE.2008.019422
38. Ma, X., Baresi, L., Ghezzi, C., Panzica La Manna, V., Lu, J.: Version-consistent dynamic reconfiguration of component-based distributed systems. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, Szeged, Hungary, pp. 245–255. ESEC/FSE '11. ACM, New York (2011). https://doi.org/10.1145/2025113.2025148
39. Ma, Z., Liu, L., Yang, H., Mylopoulos, J.: Adaptive service composition based on runtime requirements monitoring. In: Proceedings of the 2011 IEEE International Conference on Web Services, ICWS '11, Washington, DC, pp. 339–346. IEEE Computer Society, Washington, DC (2011). https://doi.org/10.1109/ICWS.2011.83
40. Mylopoulos, J.: Stateful requirements monitoring for self-repairing socio-technical systems. In: Proceedings of the 2012 IEEE 20th International Requirements Engineering Conference (RE), RE '12, Chicago, IL, pp. 121–130. IEEE Computer Society, Washington, DC (2012). https://doi.org/10.1109/RE.2012.6345796

41. Oreizy, P., Gorlick, M.M., Taylor, R.N., Heimbigner, D., Johnson, G., Medvidovic, N., Quilici, A., Rosenblum, D.S., Wolf, A.L.: An architecture-based approach to self-adaptive software. IEEE Intell. Syst. **14**(3), 54–62 (1999). https://doi.org/10.1109/5254.769885
42. Salifu, M., Yu, Y., Nuseibeh, B.: Specifying monitoring and switching problems in context. In: 15th IEEE International Requirements Engineering Conference, RE 2007, New Delhi, 15–19 Oct 2007, pp. 211–220 (2007)
43. Salifu, M., Yu, Y., Bandara, A.K., Nuseibeh, B.: Analysing monitoring and switching problems for adaptive systems. J. Syst. Softw. **85**(12), 2829–2839 (2012). http://www.sciencedirect.com/science/article/pii/S0164121212002257
44. Sasano, I., Hu, Z., Hidaka, S., Inaba, K., Kato, H., Nakano, K.: Toward bidirectionalization of ATL with GRoundTram. In: ICMT. Zurich, Switzerland LNCS, vol. 6707, pp. 138–151. Springer (2011)
45. Song, H., Huang, G., Chauvel, F., Xiong, Y., Hu, Z., Sun, Y., Mei, H.: Supporting runtime software architecture: a bidirectional-transformation-based approach. J. Syst. Softw. **84**(5), 711–723 (2011). https://doi.org/10.1016/j.jss.2010.12.009
46. Stevens, P.: Bidirectional model transformations in QVT: semantic issues and open questions. Softw. Syst. Model. **9**(1), 7–20 (2010)
47. Tisi, M., Jouault, F., Fraternali, P., Ceri, S., Bézivin, J.: On the use of higher-order model transformations. In: Model Driven Architecture – Foundations and Applications, Enschede, The Netherlands, LNCS, vol. 5562, pp. 18–33. Springer (2009). https://doi.org/10.1007/978-3-642-02674-4_3
48. Vogel, T., Giese, H.: Model-driven engineering of self-adaptive software with eurema. ACM Trans. Auton. Adapt. Syst. **8**(4), 18:1–18:33 (2014). https://doi.org/10.1145/2555612
49. Wang, Y., McIlraith, S.A., Yu, Y., Mylopoulos, J.: Monitoring and diagnosing software requirements. Autom. Softw. Eng. **16**(1), 3–35 (2009). http://link.springer.com/article/10.1007/s10515-008-0042-8
50. Wermelinger, M., Yu, Y.: Some issues in the 'archaeology' of software evolution. In: Fernandes, J.M., Lämmel, R., Visser, J., Saraiva, J. (eds.) Generative and Transformational Techniques in Software Engineering III, vol. 6491, pp. 426–445. Lecture Notes in Computer Science. Springer (2011). http://oro.open.ac.uk/22105/, International Summer School, GTTSE 2009, Braga, 6–11 July 2009, Revised Papers
51. Wermelinger, M., Yu, Y., Lozano, A., Capiluppi, A.: Assessing architectural evolution: a case study. Empir. Softw. Eng. **16**(5), 623–666 (2011). http://oro.open.ac.uk/28753/
52. Whittle, J., Sawyer, P., Bencomo, N., Cheng, B.H.C., Bruel, J.M.: RELAX: a language to address uncertainty in self-adaptive systems requirement. Requir. Eng. **15**(2), 177–196 (2010). http://link.springer.com/article/10.1007/s00766-010-0101-0
53. Yu, E.S.K.: Towards modeling and reasoning support for early-phase requirements engineering. In: Proceedings of the 3rd IEEE International Symposium on Requirements Engineering, Annapolis, MD, p. 226. RE '97. IEEE Computer Society, Washington, DC (1997). http://dl.acm.org/citation.cfm?id=827255.827807
54. Yu, Y., Lapouchnian, A., Liaskos, S., Mylopoulos, J., Leite, J.C.S.P.: From goals to high-variability software design. In: An, A., Matwin, S., Raś, Z.W., Ślęzak, D. (eds.) Foundations of Intelligent Systems, Toronto, Canada, vol. 4994, pp. 1–16. Lecture Notes in Computer Science. Springer, Berlin/Heidelberg (2008). http://link.springer.com/chapter/10.1007/978-3-540-68123-6_1
55. Yu, Y., Tun, T.T., Nuseibeh, B.: Specifying and detecting meaningful changes in programs. In: ASE, Lawrence, KS, pp. 273–282 (2011)
56. Yu, Y., Wang, Y., Mylopoulos, J., Liaskos, S., Lapouchnian, A., Prado Leite, J.C.S.D.: Reverse engineering goal models from legacy code. In: Proceedings of the 13th IEEE International Conference on Requirements Engineering, Paris, France, RE '05, pp. 363–372. IEEE Computer Society, Washington, DC (2005). https://doi.org/10.1109/RE.2005.61
57. Yu, Y., Lin, Y., Hu, Z., Hidaka, S., Hiroyuki, K., Montrieux, L.: Maintaining invariant traceability through bidirectional transformations. In: ICSE, Zurich, Switzerland, pp. 540–550 (2012)

58. Zave, P., Jackson, M.: Four dark corners of requirements engineering. ACM Trans. Softw. Eng. Methodol. **6**(1), 1–30 (1997). https://doi.org/10.1145/237432.237434
59. Zoghi, P., Shtern, M., Litoiu, M.: Designing search based adaptive systems: a quantitative approach. In: Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS '14, Hyderabad, India, pp. 7–16. ACM, New York (2014). https://doi.org/10.1145/2593929.2593935

Chapter 2
Self-Adaptation of Software Using Automatically Generated Control-Theoretical Solutions

Stepan Shevtsov, Danny Weyns, and Martina Maggio

Abstract Control theory has contributed a set of foundational techniques to handle "change" at runtime in software applications. These techniques however have fundamental limitations as well: (i) they require the development and understanding of mathematical models; (ii) synthesizing solutions is often done on a per-problem basis, discouraging flexibility and generality. Software engineering, as a discipline, has always aimed at finding reusable and modular solutions. The combination of the desire to apply formally grounded control-theoretical principles and reuse existing solutions has motivated research on the topic of *automatically generated control solutions*. This research aims at designing control strategies in an automated way from data that qualifies the given problem at hand. This chapter provides an overview of the research topic of automatically generated control-theoretical solutions, explaining the key research contributions and paving the way for future research.

2.1 Introduction

Software applications need, more than ever, to be able to deal with "change" [30, 41]. Software needs to be continuously available, which in turns requires that developers treat change as a first-class concern in the complete life cycle of the application development, operation, and maintenance. Software applications are nowadays expected to deal seamlessly with different types of change, such as resource fluctuations [37], component failures [44], requirement modifications [6, 49], different user preferences [43], and much more [1, 2, 9, 14, 27, 42]. Often,

S. Shevtsov (✉) · D. Weyns
Linnaeus University, Växjö, Sweden
KU Leuven, Leuven, Belgium
e-mail: stepan.shevtsov@lnu.se; danny.weyns@kuleuven.be

M. Maggio
Lund University, Lund, Sweden
e-mail: martina.maggio@control.lth.se

Y. Yu et al. (eds.), *Engineering Adaptive Software Systems*,
https://doi.org/10.1007/978-981-13-2185-6_2

these changes are not predictable at design time, requiring software to execute with incomplete knowledge and face new challenges during operation [50, 53]. Consequently, software engineering researchers are experimenting with new solutions that can handle change at runtime without incurring into penalties, slowdown, and downtime. Generally speaking, the software built to deal with change is often called "self-adaptive" [15, 17, 51], for the ability to modify its own behavior and adapt to the current execution conditions.

Continuous- and discrete-time control theory[1] has been identified as a promising approach to design self-adaptive software [10, 18, 26, 56]. However, the wide adoption of control-theoretical solutions in the design of self-adaptive systems has been limited by a number of factors.

First and foremost, continuous- and discrete-time control solutions often require a "physical" model of the object to be controlled. In the case of low-level resources – such as CPU, memory, and network bandwidth – researchers have proposed models that attempt to capture the phenomena of interest [3, 20, 55] with a precision sufficient to perform adaptation. However, it is very difficult to extract control-theoretical (i.e., equation-based) models for the behavior of software applications. This has been one of the main reasons why several researchers have argued that applying control theory to adapt the higher-level software elements is a more complex problem [4, 11, 22]. Other reasons are the diversity and interplay of requirements and the need for instrumenting software to obtain measurements from sensors and enacting the system through actuators [12, 28]. Second, the models often become complicated, calling for elaborate solutions from the mathematical perspective. Finally, since appropriate and accurate models are so difficult to write, existing control-based approaches are often tailored for a particular problem, while software engineers usually aim for reusable solutions. These observations have been recently confirmed by a systematic study on control-theoretical software adaptation, highlighting the shortcomings of the existing ad hoc control-theoretical solutions [47].

As a response to these shortcomings, researcher aimed at automatically generating control solutions. These solutions are general enough to tackle a variety of problems, trading off the optimality that could be reached by tailored solutions. The code for these general solutions can be automatically generated based on observations and data from the software application that should be controlled. Simple linear models describing the software behavior are automatically extracted from the data and used – at runtime – to synthesize a control solution. This chapter gives an overview of the state of the art of the research in automatically generated control strategies for software applications and outlines promising paths for future work.

The remainder of this chapter is structured as follows. Section 2.2 provides a brief background on automatically generated control-theoretical adaptation of software.

[1] In this chapter, we restrict ourselves to continuous- and discrete-time control [8, 54]. Discrete event systems are out of our scope.

In Sect. 2.3, we delve into details discussing the differences among the proposed solutions. Finally, Sect. 2.4 outlines a number of challenges for future research, and Sect. 2.5 draws some conclusions.

2.2 Background

This section explains the basic principle behind automatically generated control-theoretical solutions and its use for self-adaptation.

The overall objective of automatically generated control-theoretical adaptation is the simplification of the software design process. The aim of these strategies is to provide the software engineer with the advantages of a control-theoretical design, without the need for in-depth control expertise. The main advantage of control-theoretical solutions is the presence of *formal guarantees* [24]. If mastered correctly, the use of the knowledge coming from control theory allows for certified and verifiable solutions, where desired properties can be guaranteed *by design*. For example, with control theory it is possible to precisely calculate the amount of disturbance the system can withstand or to prove that the system will not overconsume resources in changing external conditions.

Figure 2.1 shows a typical control-theoretical feedback loop that is used in self-adaptive software systems. Reading the figure from left to right, the *Goal* represents a particular level of software quality that should be achieved by self-adaptation. The Goal is often specified as a setpoint, i.e., a certain value of a nonfunctional requirement, such as a specific service failure rate or response time. Using the setpoint and the *Measured Output* value for the same software quality, an *Error* is calculated as $Setpoint - MeasuredOutput$, where the *-1* block indicates that the Measured Output value should be subtracted. The *Feedback Controller* uses the Error in order to compute the *Control Signal*, a value or a vector of values that effect the *Software System*. If designed correctly, the Control Signal will result in a Measured Output that is equal or very close to the Goal value. The *Disturbances*, such as changing availability of resources or component failures, affect the software behavior at runtime. So one of the main purposes of control strategies is to neglect the effect of Disturbances on the system.

Historically, many manually generated control strategies used the typical feedback loop shown in Fig. 2.1. The automated strategies have two main differences

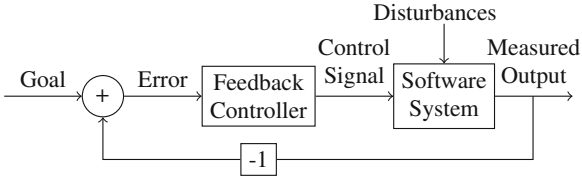

Fig. 2.1 A typical control-theoretical feedback loop

from these solutions. First, the automated strategies require certain conditions to be satisfied and the availability of specific software functions:

- The developer that wants to generate and use the control strategy should have access to the software system, which should be working and on which experiments should be done and data must be collected – the data is used in an automated way to build a model of the software that can be used for control purposes;
- The developer should be able to qualify, quantify, and measure the requirements that must be satisfied on the system – these requirements are then translated into goals and objectives that the controller will try to achieve;
- The developer must provide access to a set of sensors that get reliable data about the quantifiable objectives (e.g., measure the response times of a cloud application);
- The developer must provide access to a set of actuators (tunable parameters of the system) that can be used during runtime to modify the behavior of the software application (e.g., the percentage of rejected requests, or different implementations of the same functionality).

Second, the Feedback Controller is created automatically. Namely, the automated solution starts by running experiments on the software application, changing the values of the actuators according to predefined patterns and measuring the values of the goals in the tested configurations. With this data, the solution generates a mathematical model of the software using system identification [34].[2] Finally, this model is used to synthesize a controller that provides guarantees on certain system properties. The controller – synthesized in form of equations and subsequently in form of a code block – adapts the behavior of the software changing the values of the actuators to achieve the given goals. The resulting controller is often tunable – some parameters have default values that can be changed to alter the behavior of the controller itself. For example, parameters can be used to exploit the trade-off between robustness to disturbances and speed of convergence. The software engineer can select these parameters based on experience and on the specific execution conditions.

2.3 Automated Control-Theoretical Software Adaptation

This section outlines the research progress in self-adaptation of software using automatically generated control-theoretical solutions. We discuss five different research problems that have been explored. Figure 2.2 gives an overview of the

[2]Other model synthesis techniques can be used to produce system model. But historically, automated approaches used system identification as it is fast and approximates software well enough for controllers to work.

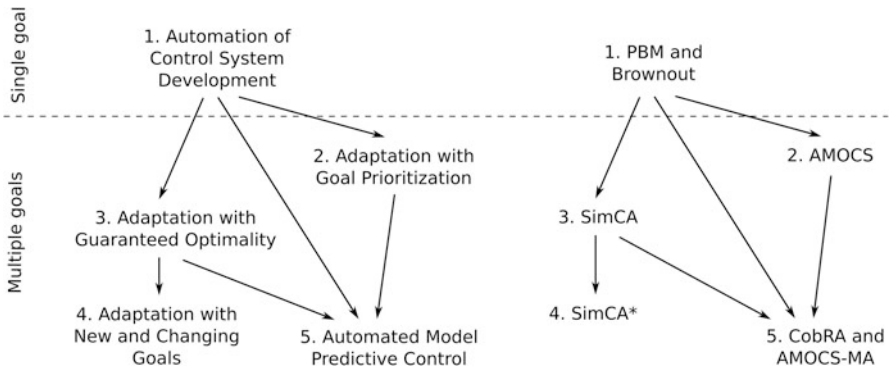

Fig. 2.2 Research in automated control-theoretical software adaptation: progress steps (left) and approaches (right)

research steps and shows representative approaches for each step. The arrows in the figure show the contribution of each step/approach to the following efforts.

The initial research was primarily targeting the automation of a control solution development. Based on prior experience with control of software applications, some generalization arose and led to the introduction of the Push-Button methodology (PBM) [22]. At the same time, a similar method called Brownout [33] was applied in a specific software domain, cloud applications. The next clear research goal has been the extension of automated methodologies to support *multiple adaptation goals* simultaneously, e.g., to achieve a specific performance level and minimize cost at the same time. The first proposed extension has been the Automated Multi-objective Control of Software (AMOCS) approach [23], followed by the Simplex Control Adaptation (SimCA) [45]. SimCA tackled the problem of multi-objective adaptation by combining controllers with the simplex optimization algorithm in a hierarchical structure. Then, SimCA* [48] introduced components that adjust the adaptation mechanism at runtime, to deal with new types of goals and changes in the set of adaptation goals (e.g., adding a new goal, removing a goal). Finally, the use of *Model Predictive Control* (MPC) was investigated. In this approach, the controller acts based on the current feedback from the software but uses the model of its own behavior to predict the software evolution. The fully automated MPC-based approach is called Automated Multi-objective Control of Software with Multiple Actuators (AMOCS-MA) [36].

The main properties of all automated control-theoretical adaptation approaches are listed in Table 2.1; these approaches will be discussed in detail in the following sections.

Table 2.1 Automated control-theoretical adaptation approaches

Approach	Inputs (goals)	Main pros	Main cons
Brownout, PBM	1-setpoint	Automation, guarantees	Handles only one goal
AMOCS	n-setpoint, 1-optimization	Multiple goals and prioritization	Suboptimal adaptation decisions
SimCA	n-setpoint, 1-optimization	Guarantees + optimality	Setpoints, needs knowledge about some of the system parameters
SimCA*	n-setpoint, n-threshold, 1-optimization	Handles new types of goals and goal changes at runtime	Needs knowledge about some of the system parameters
AMOCS-MA	n-setpoint, 1-optimization	Guarantees + optimality, does not need system knowledge, flexible computation time	Sensitive to disturbances and model inaccuracies

2.3.1 Automation of Control System Development

Control-theoretical approaches were first used in software adaptation more than a decade ago [1, 2, 14]. However, most of these approaches aim to solve a specific problem at hand. Therefore, new problems would require modifications or even replacement of a control system, which in turn requires expertise in control theory, extra resources, and effort. To overcome this concern, researchers have studied the ways to automate the entire process of control system development from the model synthesis to the formal analysis of guarantees. This became the first step of research on applying automatically generated control-theoretical solutions in software adaptation.

The representative of the first step of research are Brownout [33] and PBM [22]. Both these approaches are based on the same underlying principles (creating a first-order model from data and controlling that first-order model using pole placement). Brownout is applied to the more confined domain of cloud computing applications and is tailored to the specific problem of capacity shortages. Because of this, Brownout achieves – on its own problem – better performance than the application of the PBM controller without any modifications. We provide details on both of these approaches below.

2.3.1.1 Brownout

The main idea behind Brownout [33, 35] is to apply the principles of graceful degradation to cloud applications using control theory. Cloud applications behave according to the request-response paradigm, with clients issuing requests and a certain number of replicas of the same application providing the according responses. When producing the response to the user requests, it is often possible

to identify a part of the response that is the mandatory to display and a part of the response that would provide a better user experience and increase revenues, but is not mandatory. In the case of a travel agency website, the mandatory part of the response is the flight search, while additional optional information are car rental locations and hotel suggestions. Clearly, the application owner wants to provide the additional information, but not at the expense of losing a customer. Brownout divides the response into the two mentioned parts and measures the response time to determine how much percentages of the optional content should be served. This percentage is called the *dimmer* value. The goal of brownout is to have as big dimmer as possible, i.e., to show as much optional content as possible, without penalizing response times.

Brownout assumes that the cloud application behaves according to a simple first-order linear model, where the value of the 95th percentile of the response time τ_{95} varies depending on the dimmer value as follows:

$$\tau_{95}(k) = \alpha\,\theta(k-1) + \delta\tau_{95}(k), \tag{2.1}$$

where $\theta(k)$ is the dimmer value; $\alpha(k-1)$ is a time-varying coefficient that depends on the computing platform and can be estimated; $\delta\tau_{95}(k)$ is a disturbance, interfering with the nominal system's behavior; and k is the discrete time instance.

Based on the model (2.1), the following controller is then synthesized using loop shaping [8]:

$$\theta^*(k) = \theta(k-1) + \frac{1-p_b}{\hat{\alpha}(k)} \cdot e_{\tau 95}(k) \tag{2.2}$$

where $\hat{\alpha}(k)$ is an estimate of $\alpha(k)$ obtained with a recursive least squares (RLS) filter, p_b is a controller parameter called pole, and $e_{\tau 95}(k)$ is the error between the desired 95th percentile of the response time $\bar{\tau}_{95}(k)$ and the actual value. The pole p_b can be used to trade the speed of controller convergence for robustness to model perturbations. The analysis of the brownout closed loop allows to prove a number of properties, such as system stability and zero steady-state error. However, this proof is subject to how well the model (2.1) approximates the behavior of the cloud application.

Brownout uses a single actuator (the dimmer value) to achieve a single goal, specified in terms of a setpoint for the response time statistic. The control strategy in Brownout can be greatly improved, and many follow-ups were devised. For example, an event-based version of the brownout paradigm [19] explores a similar cloud problem but controls the server queue length. Furthermore, extensions that include brownout load balancing were considered [21, 32]. They demonstrate that state-of-the-art load balancers which use response times as a measure for determining where to send requests do not work with brownout-aware applications. This is a natural limitation as the brownout controller can satisfy only a single goal and therefore cannot form a multi-objective control strategy with other controllers.

Brownout was designed specifically for cloud applications, so strictly speaking, it is not a generally applicable solution. However, it is important to include Brownout in this work as it became the first building block for development of automated control-theoretical adaptation. The generally applicable Push-Button methodology, discussed in the following section, is based on the same principles and shares many elements with Brownout.

2.3.1.2 Push-Button Methodology

The PBM methodology [22] works in a way similar to Brownout but *goes beyond a single goal and a single actuator. Also, it introduces the idea of identifying the model online. Unlike in Brownout, where model is pre-determined, PBM builds a model directly from the data received by running experiments on the software and produces a controller for this model*. Figure 2.3 shows the two phases of the methodology: model building and controlling.

The input required by PBM from a software engineer is a method to set the actuator value and a method to collect measurements about the system goal. Based on this input, PBM first produces a linear model $\mathcal{M}$ of the software:

$$\mathcal{M}: \; y(k) = \alpha(k-1) \cdot u(k-1) \tag{2.3}$$

where the input u is the value of the actuator, the output y is the effect of the actuator on the goal, the parameter α is a time-varying coefficient that is determined during model building by feeding different input values as u and measuring the resulting outputs y, and k is a discrete time instance.

After the model building, the controller synthesis phase automatically generates a proportional-integral controller C that works on the model $\mathcal{M}$ and adapts the software.

$$C: \; u(k) = u(k-1) + \frac{1-p_b}{\alpha} \cdot e(k) \tag{2.4}$$

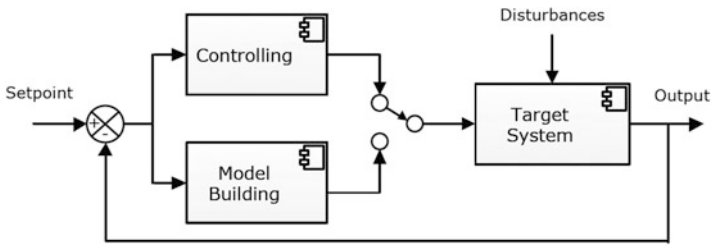

Fig. 2.3 The two operational phases of PBM

The controller has one parameter, p_b, that has the same role that it had in the Brownout controller. More guidelines on how to tune the controller parameter p_b are available in [22].

To address model inaccuracies and small perturbations during software operation, the value of α is updated at runtime. In case of critical changes (e.g., a software component failure), PBM restarts the model building phase and regenerates the controller.

2.3.2 Adaptation with Goal Prioritization

In order to automatically create control solutions for more practical problems, researches have studied the ways to address multiple adaptation goals simultaneously. The first automated approach that offered control-based multi-objective software adaptation was AMOCS [23]. *This approach extends the methodology behind PBM to use multiple actuators and multiple controllers in a cascaded structure*; see Fig. 2.4.

AMOCS works as follows. The set of available actuators $\mathcal{A} = \{a_1, \ldots, a_m\}$ is partitioned to reach the set of goals $\mathcal{G} = \{g_1, \ldots, g_n\}$, where $m \geq n$, i.e., the system should have more actuators than goals. The goals are added into the set $\mathcal{G}$ according to their priority order, forming the chain $< g_1, g_2, \ldots g_n >$, where g_1 is the most important goal and g_n is the least important one. All goals, except the last one, are specified as setpoint values to be achieved by the adaptation. The last goal g_n is always the optimization of a specific value (e.g., maximization of profit, minimization of cost). $\mathcal{A}_i$ denotes the subset of actuators used to achieve the goal g_i. AMOCS assumes that every actuator is used:

$$\bigcup_{i \in \{1 \ldots n\}} \mathcal{A}_i = \mathcal{A}, \tag{2.5}$$

and each actuator is assigned to a single goal only:

$$\forall i, j \in \{1 \ldots n\}, i \neq j \implies \mathcal{A}_i \cap \mathcal{A}_j = \emptyset, \tag{2.6}$$

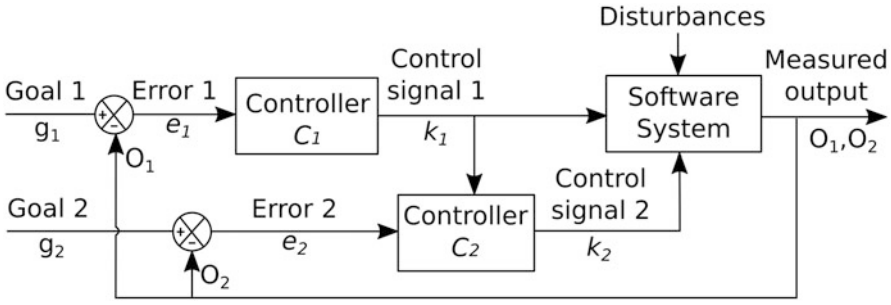

Fig. 2.4 A self-adaptive software with AMOCS (for 2 goals)

A first instance of PBM controller C_1, see (2.4) for a controller description, is then used to translate the discrete set of configurations of all the actuators $\mathcal{A}_1$ related to the first goal g_1 into a single configuration that satisfies this goal. This configuration is then sent in the form of control signal k_1 to the software system and to the second instance of PBM controller C_2,which tries to achieve the second goal g_2 with the available actuators $\mathcal{A}_2$ and operating conditions. The resulting configuration is sent to software as control signal k_2. If goals g_1 and g_2 are not related, the control signal k_1 will still be received by controller C_2, but it will not affect the reachability of the goal g_2.

In this controller chain, only the first goal is guaranteed to be stable, while the stability of the others depend on the disturbances and on the control values set by the previous controllers in the chain. In other words, the goal g_2 is guaranteed to be reached only if control signal k_1 allows to reach it. The last optimization requirement is reached to the best of the chain ability; hence there is no guarantee for the solution optimality. Despite the lack of formal guarantees, the experiments with AMOCS show that the chain of controllers behaves well in a variety of different scenarios and can successfully handle multiple goals of a setpoint type.

2.3.3 Adaptation with Guaranteed Optimality

Guided by the need for stronger adaptation guarantees in systems with multiple goals, the research explored new ways to automatically build the control system. The approach resulting from these efforts is called Simplex Control Adaptation (SimCA) [45]. *SimCA combines PBM with the simplex optimization method, utilizing the advantages of both approaches*. SimCA finds a system configuration that satisfies multiple goals, reaches optimality with respect to an additional goal, achieves robustness to environmental disturbances and measurement inaccuracy, and provides control-theoretical adaptation guarantees. To that end, SimCA runs on-the-fly experiments on the software in an automated fashion, builds a set of linear models of the software at runtime, creates a set of tunable PI controllers that operate on these models and independently compute control signals for each of the goals, and combines controller outputs using the simplex method to adapt the system. Figure 2.5 schematically shows the primary building blocks of SimCA.

SimCA builds a self-adaptive system in three phases executed during system operation:

1. In the *Identification* phase, n linear models of the controlled system are built. SimCA uses multiple instances of the PBM model $\mathcal{M}$, where each model $\mathcal{M}_i, i \in [1, n]$, is responsible for one goal s_i. Similar to PBM, each model is automatically learned at runtime by running the experiments on the software (see Sect. 2.3.1 for details). As in PBM, the model $\mathcal{M}_i$ automatically adjusts at runtime according to changes in the system behavior.

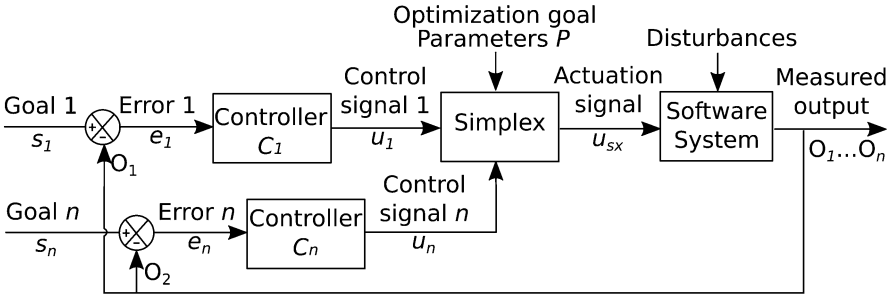

Fig. 2.5 A self-adaptive software with SimCA

2. In the *Controller Synthesis* phase, SimCA constructs a set of n controllers; each controller C_i is responsible for the i-th goal. C_i calculates the control signal $u_i(k)$ at the current time step k depending on the previous value of control signal $u_i(k-1)$, model coefficient α_i, parameter pole p_i, and the error $e_i(k-1)$, with $e_i = s_i - O_i$. Similar to PBM, p_i is used to tune the controllers and trade off different system properties.

$$u_i(k) = u_i(k-1) + \frac{1-p_i}{\alpha_i} \cdot e_i(k-1) \qquad (C_i)$$

3. In the *Operation* phase, the set of controllers effectively perform control. Each controller C_i manages one goal s_i, rejects disturbances acting on the according output $O_i(k)$, and provides an output signal $u_i(k)$. SimCA combines the signals $u_i(k)$ from all the controllers and uses the simplex method to calculate the actuation signal u_{sx} that drives the system toward an output that satisfies all adaptation goals.

   Generally, the simplex method allows to find an optimal solution to a linear problem written in the standard form:

$$\max\{c^{\mathrm{T}}x \mid Ax \leq b; x \geq 0\} \qquad (2.7)$$

   where x represents the vector of variables (to be determined), c and b are vectors of (known) coefficients, A is a (known) matrix of coefficients, and $(\cdot)^{\mathrm{T}}$ is the matrix transpose [16].

   In SimCA each equation, except the last one, represents a goal s_i to be satisfied. The last equation ensures that the system selects a valid actuation signal by constraining the values that can be taken by elements of the vector x, e.g., $x \geq 0$. The control signals $u_i(k)$ produced during the control phase replace constants b, whereas matrix A and vector c^{T} are substituted with the monitored parameters $\mathcal{P}(k)$ of the system. The goal of simplex is to find a proper actuation signal u_{sx}, i.e., vector x.

Note that SimCA uses a simplex variant with equalities ($Ax = b$) in order to prevent simplex from changing the effect of control signal $u_i(k)$ on the output signal $O_i(k)$. Instead, simplex is responsible for seamless translation of control signals $u_i(k)$ to actuation signal u_{sx}. This allows to provide the entire set of *control-theoretical guarantees*, including stability, absence of overshoot, tunable settling time, and robustness to disturbances. A major advantage of SimCA over approaches from the previous research steps is that simplex guarantees solution optimality, meaning that all the system goals are guaranteed to be achieved. An interested reader may refer to [45] for further details.

A follow-up work [46] compares SimCA with an architecture-based ActivFORMS approach using a simulated service-based system. The study shows that both approaches can deal with multiple goals and provide guaranteed solution optimality. However, SimCA achieves better results in the presence of runtime changes as it does not rely on data verified at design time. Except optimality, the two adaptation approaches offer different guarantees. The design of SimCA adaptation mechanism allows to formally prove the properties of underlying system and guarantee that they will hold at runtime independent of the system parameters. ActivFORMS, on the other hand, can guarantee the functional correctness of the implementation of the adaptation algorithm, such as the absence of erroneous states and correct interaction between adaptation components.

2.3.4 Adaptation with New and Changing Goals

One interesting research line for automated methodologies and for control methodologies in general is the selection and support of types of adaptation goals. The previously developed automated approaches had two major drawbacks. First, they addressed goals specified either in the form of particular setpoint values to be achieved by the system (S-goal) or values to be optimized (O-goal), while many software systems need to address a threshold goal that keeps a value above/below a threshold (T-goal). A typical example is limiting the response time of a Web server. Approaches such as described in [31, 33, 38] solve this problem either by optimizing the response time (O-goal) or by defining a setpoint for response time that the controller should guarantee (S-goal), when the actual requirement is to keep response time lower than a certain threshold. Second, the previously developed approaches did not provide support for changing the set of system requirements during operation, which requires on-the-fly adjusting, activation, and deactivation of adaptation goals. Changing requirements are important in practice, e.g., to deal with drastic changes in the system or its environment that may require the system to change from one set of requirements to another.

In order to address the two mentioned concerns, the SimCA approach (see Sect. 2.3.4) was reworked and upgraded into SimCA* [48]. *Compared to original*

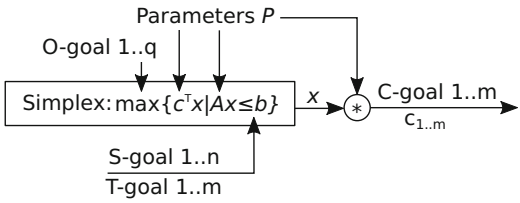

Fig. 2.6 Goal Transformation phase of SimCA*

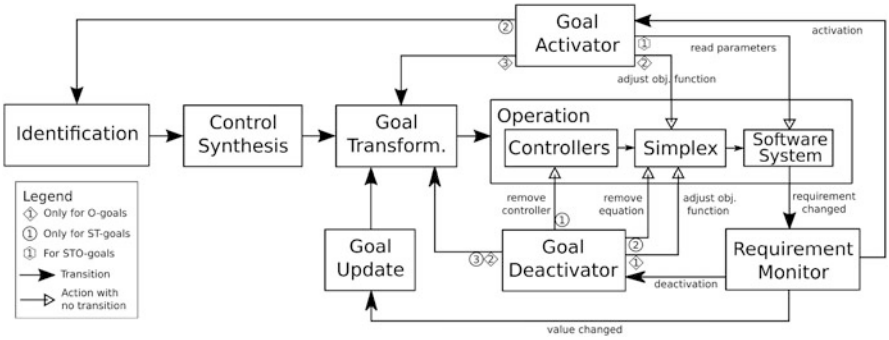

Fig. 2.7 Dealing with requirement changes in SimCA*. Numbers in circles/diamonds show the sequence of actions

SimCA, the new approach includes an additional Goal Transformation phase (Fig. 2.6) *and the necessary mechanisms to support changing system requirements by activating/deactivating goals* (Fig. 2.7).

The *Goal Transformation phase* of SimCA* is performed between the Controller Synthesis and Operation phases. The purpose of this phase is to transform T-goals into goals that can be controlled by the original SimCA controller (C_i). As such, the approach uses simplex, where each equation in the system (2.7), except the last one, represents an S-goal or T-goal to be satisfied (see Fig. 2.6). Equalities are used for S-goals, while inequalities are used for T-goals. The last equation ensures that the system selects a valid solution, the vector x, by the means of constraints, e.g. $x \geq 0$. The goal of simplex is to find such vector x that satisfies all system goals; the details of how simplex finds such a solution can be found in the linear programming literature [16]. Knowing the vector x, each T-goal is transformed into a controller goal (C-goal) c_i as follows: $c_i = \mathcal{P}_i(k) * x$. The resulting C-goal represents a particular value of a corresponding T-goal. For example, a T-goal that should keep a value below a threshold will be transformed into a C-goal with a value that is equal to the lowest possible value of the goal below that threshold that satisfies all other requirements. All the C-goals and the original S-goals are then used by controllers (C_i) in the usual Operation phase described in Sect. 2.3.4.

In order to address the *changing system requirements*, SimCA* is equipped with a Requirement Monitor, Goal Activator, and Goal Deactivator components; see Fig. 2.7. The Requirement Monitor triggers the corresponding adaptation component after any system requirement is changed. The Goal Activator first reads

the relevant parameters $\mathcal{P}$ related to the activated goal. Then, in case of O-goal activation, it inserts $\mathcal{P}$ into the objective function c^{T} of simplex, performs a Goal Transformation (described above), and proceeds to standard Operation phase. In case of S- or T-goal, the Goal Activator triggers a standard Identification phase for the new goal, which is followed by Controller Synthesis, Goal Transformation, and Operation. The Goal Deactivator removes the according elements of the adaptation mechanism. Namely, when an S- or T-goal is deactivated, the corresponding controller is removed together with the equation responsible for the goal being deactivated. When an O-req is deactivated, the corresponding variables are removed from the objective function c^{T} of simplex. After that, the Goal Deactivator always triggers a Goal Transformation adapting the configuration of the control system to the new set of requirements, after which the system returns to standard Operation.

2.3.5 Automated Model Predictive Control

The scope of applicability of the first multi-objective control solutions is limited in different ways. For example, SimCA cannot prioritize goals or use infinite sets of values for the actuators, while AMOCS produces suboptimal solutions. *To eliminate these limitations, researchers have studied the application of automated model predictive control (MPC)* – a technique based on the optimization of a cost function and on the prediction of a future outcome of the adaptation. Generally, in control theory, MPC is considered particularly well suited for multi-objective problems with optimization, because all the interdependencies between actuators and goals are taken into account simultaneously, achieving a truly optimal solution.

The first research effort that identifies automated MPC as a potential multi-objective control strategy for self-adaptive systems is [5]. However, it lacks details and does not provide any analysis of guarantees. In the same research line – again for a specific problem, but with a general overlook – CobRA [7] provides a framework to reason about MPC and its application to computing systems. Although the model in CobRA has to be generated manually and fed to the system, the solution of the MPC problem is general with respect to the involved quantities. The paper only provides an example of the framework application, which also requires extensive manual tuning in order to tailor the equations to a specific problem. Although formal guarantees are not discussed in CobRA, it is possible to prove that they hold to the extent that the model allows. PLA [38, 39] is based on similar principles that CobRA. It uses a model of the environment and of the software to determine the best strategy to be followed using a model checker with the ability of looking into the future expectations for the system. CobRA and PLA have been compared [40] showing similar results but a different runtime behavior. The authors conclude that the concrete approach should be picked based on the problem at hand. For example, CobRA suits more for continuous inputs, while PLA works better with discrete control.

Finally, a fully automated model predictive control strategy was developed as a part of AMOCS-MA approach [36]. Similar to other automated solutions, AMOCS-MA starts with a model building phase. The following model S is synthesized:

$$S = \begin{cases} x(k+1) = A \cdot x(k) + B \cdot \Delta a(k) \\ O(k) = C \cdot x(k) \end{cases} \tag{2.8}$$

where k is a discrete time instance, $O(k)$ is the vector of all system outputs at time k, $\Delta a(k)$ is the control signal containing values of all actuators, $x(k)$ is the current system state, $x(k+1)$ is the next system state, and A, B, and C are the matrices of coefficients obtained with model learning by running experiments on the software at runtime. One of the AMOCS-MA advantages is that it reduces the model learning time by using special input signals in the model building phase; see details in [36]. As in other automated approaches, the model S is updated according to runtime changes that appear in the software system.

The model S is used by an MPC controller to minimize the following cost function, which handles all S-goals and O-goals:

Minimize $\Delta a(k+i-1)$, with $i = 1 \ldots L$ in:

$$\sum_{i=1}^{L} \left\langle \sum_{j=1}^{p} q_j \cdot [O_j(k+i) - g_j(k+i)]^2 + \sum_{l=1}^{m} r_l \cdot \Delta a_l(k+i-1)^2 \right\rangle \tag{2.9}$$

Subject to: model S (2.8) and additional $\Delta a(k)$ constraints (see [36])

where k is a discrete time instance, L is the number of discrete time instances in the future used for predicting software behavior, p is the number of goals, q_j is the weight of goal j (allows goal prioritization), $O_j(k+i)$ is the predicted measured output of goal j at the i-th step in future, $g_j(k+i)$ is the value of goal j at the i-th step in future (this value is constant if goals do not change at runtime), m is the number of actuators, r_l is the weight of actuator l (allows actuator prioritization), and $\Delta a_l(k+i-1)$ is the predicted change in the value of actuator l at the $i-1$-th step in future.

As the controller depends on the model (2.8), it requires information about the system state $x(k)$. However, it is problematic to measure the system state $x(k)$ directly, so it is estimated instead. To accomplish this, AMOCS-MA uses a Kalman filter that computes an estimate $\hat{x}(k)$ of the state $x(k)$ based on the previous control signal $\Delta a(k-1)$, the measured outputs $O_j(k)$, prediction error, and a number of other parameters.

Using the estimate $\hat{x}(k)$, the MPC controller solves (2.9) and produces an optimal plan of control actions for the future i steps: $\Delta a(k+i-1)$, with $i = 1 \ldots L$. The plan $\Delta a(k+i-1)$ contains particular values of all actuators at time instance $(k+i-1)$. However, AMOCS-MA uses only the first action of the plan, i.e., $\Delta a(k)$ is applied to software; see Fig. 2.8.

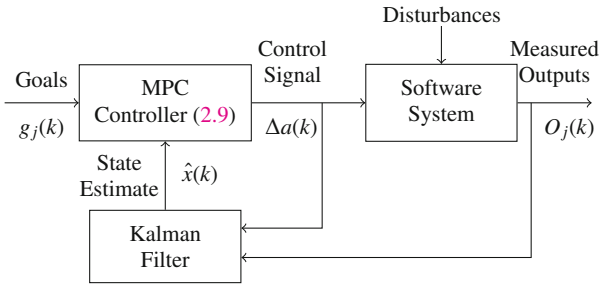

Fig. 2.8 A self-adaptive system with AMOCS-MA

The controller (2.9) guarantees stability, zero steady-state error, and minimal settling time by design. It also guarantees the optimality of a cost function specified by the user. This function has tunable weights for the system goals q_j and actuators r_l, allowing to trade off different system properties, e.g., to prioritize response time over cost.

2.4 Challenges

The analysis of automated control-theoretical adaptation solutions showed the use of various controllers, from hierarchical adaptive PI control (SimCA) to model predictive control (AMOCS-MA). However, most of these approaches use the PBM model (2.3) or its variations. Indeed, one of the key points behind this line of research is the difficulty in finding generic models that describe software applications and their behavior. Although the usual software models – architectural models and UML descriptions – are a very good reference to understand how the control code interfaces with the rest of the software application, they are not suitable for the control design process. To design a controller, there is usually a need to understand how the quantities that should be controlled are influenced by the actuators that one has available. Depending on the modeling effort that the software engineer is willing to do, the control strategies can be more or less effective:

- PLA [38, 39] and Brownout [33], for example, use explicit modeling of both the software behavior and the environment. Explicit modeling goes a long way for improving the performance of the control strategy that can be perfectly tailored for a new scenario using the given knowledge. Generally speaking, when an explicit model is available, the spectrum of results that it is possible to obtain is much wider, opening up possibilities and allowing for more precise results.
- SimCA [45] and SimCA* [48] lift some of the requirements on the modeling side. While no explicit disturbance model is written, the system parameters specified in the Simplex algorithm are part of prior knowledge that is given to the control strategy and that the controller does not have to identify based on experiments.

- The PBM [22], AMOCS [23], and AMOCS-MA [36] approaches use implicit modeling requiring a very limited effort from the software engineer. The engineer should only specify the actuators and sensor and possibly some weights that are unrelated to the model itself but specify the properties of controller and how to reach the goals. Despite the lack of modeling needs from the software engineer, these approaches still build a representation of the software in the form of equations in their model building phase. The synthesized model is then used to create a controller.
- Advances in control theory have recently unveiled a new set of methods denoted model-free control [13, 25, 29]. Model-free control synthesis does not build a model of the system to be controlled but only uses data to optimize a control strategy. To date, model-free control has not been applied to software and could open possibilities for performance improvement and to tackle the complexity of software systems in an automated way.

Apart from using the same type of model, all the automated approaches discussed in this chapter synthesize centralized control solutions deployed on a single software product. Such approaches are not suitable for systems where communication between components is limited or very costly. A recent work on architecture-based adaptation [52] introduced a number of patterns for designing decentralized adaptation solutions, where controllers make independent decisions but have some kind of interaction. The automated control solutions may definitely benefit from this and similar efforts, as they provide means to adapt an entirely new class of software systems.

2.5 Conclusions

Throughout the recent years, the automatically generated control-theoretical solutions have made a huge progress. Starting from addressing a single adaptation requirement, these solutions can now handle multiple goals of different types, deal with addition or removal of system requirements on the fly, or even adapt based on the predicted software evolutions. In this chapter, we listed the key research steps that led to such progress and highlighted the main approaches representing each of the steps. Surely, the automated approaches have limitations. For example, they use simple models that are not always accurate, and they are less effective in specific scenarios than controllers finely tuned for those scenarios. However, the main advantage of automated control comes from these limitations: simple models in combination with a generally applicable controller allow to build a control-based self-adaptive system without involvement of a control expert.

As for the future of automated control-based solutions, the research efforts can be aimed in two directions. First, as the scope of applicability and practical effectiveness of existing solutions is often unclear, these solutions should be tested in the industrial settings. Second, the researchers could use more state-of-the-art practices, such as model-free control or decentralized adaptation.

References

1. Abdelwahed, S., Kandasamy, N., Neema, S.: Online control for self-management in computing systems. In: Proceedings. RTAS 2004. 10th IEEE Real-Time and Embedded Technology and Applications Symposium, Toronto, pp. 368–375 (2004)
2. Abdelzaher, T.F., Shin, K.G., Bhatti, N.: Performance guarantees for web server end-systems: a control-theoretical approach. IEEE Trans. Parallel Distrib. Syst. **13**(1), 80–96 (2002)
3. Abdelzaher, T., Stankovic, J., Lu, C., Zhang, R., Lu, Y.: Feedback performance control in software services. IEEE Control Syst. **23**(3), 74–90 (2003)
4. Andersson, J., de Lemos, R., Malek, S., Weyns, D.: Modeling dimensions of self-adaptive software systems. In: Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J. (eds.) Software Engineering for Self-Adaptive Systems, pp. 27–47. Springer Berlin/Heidelberg (2009). https://doi.org/10.1007/978-3-642-02161-9_2
5. Angelopoulos, K., Papadopoulos, A.V., Mylopoulos, J.: Adaptive predictive control for software systems. In: Proceedings of the 1st International Workshop on Control Theory for Software Engineering, CTSE 2015, Bergamo, pp. 17–21. ACM (2015)
6. Angelopoulos, K., Souza, V.E.S., Mylopoulos, J.: Capturing variability in adaptation spaces: a three-peaks approach. In: Johannesson, P., Lee, M.L., Liddle, S.W., Opdahl, A.L., Pastor López, Ó. (eds.) Proceedings of Conceptual Modeling: 34th International Conference, ER 2015, Stockholm, 19–22 Oct 2015, pp. 384–398. Springer International Publishing, Cham (2015). https://doi.org/10.1007/978-3-319-25264-3_28
7. Angelopoulos, K., Papadopoulos, A.V., Silva Souza, V.E., Mylopoulos, J.: Model predictive control for software systems with cobra. In: Proceedings of the 11th International Symposium on Software Engineering for Adaptive and Self-Managing Systems. SEAMS '16, Austin, pp. 35–46. ACM, New York (2016). https://doi.org/10.1145/2897053.2897054
8. Åström, K.J., Murray, R.M.: Feedback Systems: An Introduction for Scientists and Engineers. Princeton University Press, Princeton (2008)
9. Babu, S.: Towards automatic optimization of mapreduce programs. In: SoCC, pp. 137–142. ACM, New York (2010)
10. Brun, Y., et al.: Engineering self-adaptive systems through feedback loops. Lect. Notes Comput. Sci. **5525**, 48–70 (2009)
11. Brun, Y., Desmarais, R., Geihs, K., Litoiu, M., Lopes, A., Shaw, M., Smit, M.: A design space for self-adaptive systems. In: de Lemos, R., Giese, H., Müller, H.A., Shaw, M. (eds.) Software Engineering for Self-Adaptive Systems II: International Seminar, Dagstuhl Castle, 24–29 Oct 2010 Revised Selected and Invited Papers, pp. 33–50. Springer, Berlin/Heidelberg (2013). https://doi.org/10.1007/978-3-642-35813-5_2
12. Cai, K.Y., Cangussu, J., DeCarlo, R.A., Mathur, A.: An overview of software cybernetics. In: Eleventh Annual International Workshop on Software Technology and Engineering Practice, Amsterdam, pp. 77–86 (2003)
13. Campi, M.C., Savaresi, S.M.: Direct nonlinear control design: the virtual reference feedback tuning (VRFT) approach. IEEE Trans. Autom. Control **51**(1), 14–27 (2006)
14. Cangussu, J.A.W., Cooper, K., Li, C.: A control theory based framework for dynamic adaptable systems. In: Proceedings of the 2004 ACM Symposium on Applied Computing SAC '04, Nicosia, pp. 1546–1553. ACM, New York (2004). https://doi.org/10.1145/967900.968209
15. Cheng, B.H., et al.: Software engineering for self-adaptive systems: a research roadmap. In: Cheng, B.H.C. (eds.) Software Engineering for Self-Adaptive Systems, LNCS, vol. 5525. Springer, Berlin/New York (2009)
16. Dantzig, G.B., Thapa, M.N.: Linear Programming 1: Introduction. Springer, New York (1997)
17. de Lemos, R., et al.: Software engineering for self-adaptive systems: a second research roadmap. In: de Lemos, R., Giese, H., Müller, H.A., Shaw, M. (eds.) Software Engineering for Self-Adaptive Systems II. Lecture Notes in Computer Science, vol. 7475. Springer, Berlin/Heidelberg (2013)

18. de Lemos, R., Garlan, D., Giese, H.: Software Engineering for Self-Adaptive Systems: Assurances, Dagstuhl Seminar 13511 (2013)
19. Desmeurs, D., Klein, C., Papadopoulos, A., Tordsson, J.: Event-driven application brownout: reconciling high utilization and low tail response times. In: 2015 International Conference on Cloud and Autonomic Computing (ICCAC), Cambridge, pp. 1–12 (2015)
20. Diao, Y., Gandhi, N., Hellerstein, J., Parekh, S., Tilbury, D.: Using MIMO feedback control to enforce policies for interrelated metrics with application to the apache web server. In: NOMS 2002. 2002 IEEE/IFIP Network Operations and Management Symposium, Florence, pp. 219–234 (2002)
21. Durango, J., Dellkrantz, M., Maggio, M., Klein, C., Papadopoulos, A., Hernandez-Rodriguez, F., Elmroth, E., Arzen, K.E.: Control-theoretical load-balancing for cloud applications with brownout. In: 2014 IEEE 53rd Annual Conference on Decision and Control (CDC), Los Angeles, pp. 5320–5327 (2014)
22. Filieri, A., Hoffmann, H., Maggio, M.: Automated design of self-adaptive software with control-theoretical formal guarantees. In: Proceedings of the 36th International Conference on Software Engineering, ICSE 2014, Hyderabad, pp. 299–310. ACM (2014)
23. Filieri, A., Hoffmann, H., Maggio, M.: Automated multi-objective control for self-adaptive software design. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, pp. 13–24. ACM, New York (2015). https://doi.org/10.1145/2786805.2786833
24. Filieri, A., Maggio, M., Angelopoulos, K., D'Ippolito, N., Gerostathopoulos, I., Hempel, A., Hoffmann, H., Jamshidi, P., Kalyvianaki, E., Klein, C., Krikava, F., Misailovic, S., Papadopoulos, Alessandro, V., Ray, S., Sharifloo, Amir, M., Shevtsov, S., Ujma, M., Vogel, T.: Software engineering meets control theory. In: Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, Florence (2015). https://hal.inria.fr/hal-01119461
25. Fliess, M., Join, C.: Model-free control. Int. J. Control. **86**(12), 2228–2252 (2013)
26. Hellerstein, J.L., Diao, Y., Parekh, S., Tilbury, D.M.: Feedback Control of Computing Systems. Wiley, New York (2004)
27. Herodotou, H., Babu, S.: Profiling, what-if analysis, and cost-based optimization of mapreduce programs. PVLDB **4**(11), 1111–1122 (2011)
28. Hoffmann, H., Eastep, J., Santambrogio, M.D., Miller, J.E., Agarwal, A.: Application heartbeats: A generic interface for specifying program performance and goals in autonomous computing environments. In: Proceedings of the 7th International Conference on Autonomic Computing, ICAC '10, Reston, pp. 79–88. ACM, New York (2010). https://doi.org/10.1145/1809049.1809065
29. Hou, Z., Jin, S.: Data-driven model-free adaptive control for a class of MIMO nonlinear discrete-time systems. IEEE Trans. Neural Netw. **22**(12), 2173–2188 (2011)
30. Kephart, J., Chess, D.: The vision of autonomic computing. Computer **36**(1), 41–50 (2003)
31. Kihl, M., Robertsson, A., Wittenmark, B.: Performance modelling and control of server systems using non-linear control theory. In: J. Charzinski, R.L., Tran-Gia, P. (eds.) Providing Quality of Service in Heterogeneous Environments Proceedings of the 18th International Teletraffic Congress – ITC-18, Teletraffic Science and Engineering, vol. 5, pp. 1151–1160. Elsevier (2003). http://www.sciencedirect.com/science/article/pii/S1388343703802640
32. Klein, C., Papadopoulos, A., Dellkrantz, M., Durango, J., Maggio, M., Arzen, K.E., Hernandez-Rodriguez, F., Elmroth, E.: Improving cloud service resilience using brownout-aware load-balancing. In: 2014 IEEE 33rd International Symposium on Reliable Distributed Systems (SRDS), Nara, pp. 31–40 (2014)
33. Klein, C., Maggio, M., Årzén, K.E., Hernández-Rodriguez, F.: Brownout: building more robust cloud applications. In: Proceedings of the 36th International Conference on Software Engineering, Hyderabad, pp. 700–711. ICSE 2014. ACM (2014)
34. Ljung, L.: System Identification: Theory for the User. Prentice-Hall, Inc., Upper Saddle River (1986)

35. Maggio, M., Klein, C., Årzén, K.E.: Control strategies for predictable brownouts in cloud computing. In: IFAC Proceedings Volumes, vol. 47, pp. 689–694 (2014)
36. Maggio, M., Papadopoulos, A.V., Filieri, A., Hoffmann, H.: Automated control of multiple software goals using multiple actuators. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. ESEC/FSE 2017, Paderborn, pp. 373–384. ACM, New York (2017). https://doi.org/10.1145/3106237.3106247
37. Mars, J., Tang, L., Hundt, R., Skadron, K., Soffa, M.L.: Bubble-up: increasing utilization in modern warehouse scale computers via sensible co-locations. In: Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO-44, Porto Alegre, pp. 248–259. ACM, New York (2011). https://doi.org/10.1145/2155620.2155650
38. Moreno, G.A., Cámara, J., Garlan, D., Schmerl, B.: Proactive self-adaptation under uncertainty: a probabilistic model checking approach. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. ESEC/FSE 2015, Bergamo, pp. 1–12. ACM, New York (2015). https://doi.org/10.1145/2786805.2786853
39. Moreno, G.A., Cámara, J., Garlan, D., Schmerl, B.: Efficient decision-making under uncertainty for proactive self-adaptation. In: 2016 IEEE International Conference on Autonomic Computing (ICAC), Würzburg, pp. 147–156. IEEE (2016)
40. Moreno, G.A., Papadopoulos, A.V., Angelopoulos, K., Cámara, J., Schmerl, B.: Comparing model-based predictive approaches to self-adaptation: CobRA and PLA. In: Proceedings of the 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems. SEAMS '17, Buenos Aires, pp. 42–53. IEEE Press, Piscataway (2017). https://doi.org/10.1109/SEAMS.2017.2
41. Oreizy, P., Medvidovic, N., Taylor, R.N.: Runtime software adaptation: framework, approaches, and styles. In: Companion of the 30th International Conference on Software Engineering. ICSE Companion '08, Leipzig, pp. 899–910. ACM, New York (2008). https://doi.org/10.1145/1370175.1370181
42. Rizvandi, N., Taheri, J., Zomaya, A.: On using pattern matching algorithms in mapreduce applications. In: ISPA, pp. 75–80 (2011)
43. Sayyad, A.S., Menzies, T., Ammar, H.: On the value of user preferences in search-based software engineering: a case study in software product lines. In: Proceedings of the 2013 International Conference on Software Engineering. ICSE '13, San Francisco, pp. 492–501. IEEE Press, Piscataway (2013). http://dl.acm.org/citation.cfm?id=2486788.2486853
44. Schwarzkopf, M., Konwinski, A., Abd-El-Malek, M., Wilkes, J.: Omega: flexible, scalable schedulers for large compute clusters. In: Proceedings of the 8th ACM European Conference on Computer Systems. EuroSys '13, Prague, pp. 351–364. ACM, New York (2013). https://doi.org/10.1145/2465351.2465386
45. Shevtsov, S., Weyns, D.: Keep it simplex: satisfying multiple goals with guarantees in control-based self-adaptive systems. In: 24th ACM SIGSOFT International Symposium on the Foundations of Software Engineering. FSE 2016, Seattle (2016)
46. Shevtsov, S., Iftikhar, M.U., Weyns, D.: SimCA vs ActivFORMS: comparing control- and architecture-based adaptation on the TAS exemplar. In: Proceedings of the 1st International Workshop on Control Theory for Software Engineering, CTSE 2015, Bergamo, pp. 1–8. ACM, New York (2015). https://doi.org/10.1145/2804337.2804338
47. Shevtsov, S., Berekmeri, M., Weyns, D., Maggio, M.: Control-theoretical software adaptation: a systematic literature review. IEEE Trans. Softw. Eng. **44**(8), 784–810 (2018)
48. Shevtsov, S., Weyns, D., Maggio, M.: Handling new and changing requirements with guarantees in self-adaptive systems using SimCA. In: Proceedings of the 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems. SEAMS '17, Buenos Aires, pp. 12–23. IEEE Press, Piscataway (2017). https://doi.org/10.1109/SEAMS.2017.3
49. Silva Souza, V.E., Lapouchnian, A., Robinson, W.N., Mylopoulos, J.: Awareness requirements for adaptive systems. In: Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems. SEAMS '11, Waikiki, pp. 60–69. ACM, New York (2011). https://doi.org/10.1145/1988008.1988018

50. Souza, V.E.S., Lapouchnian, A., Mylopoulos, J.: (Requirement) evolution requirements for adaptive systems. In: Proceedings of the 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems. SEAMS '12, Zurich, pp. 155–164. IEEE Press, Piscataway (2012). http://dl.acm.org/citation.cfm?id=2666795.2666820
51. Weyns, D.: Software engineering of self-adaptive systems: an organised tour and future challenges. In: Cha, S., Taylor, R.N., Kang, K.C. (eds.) Handbook of Software Engineering. Springer, Cham (2018)
52. Weyns, D., Schmerl, B., Grassi, V., Malek, S., Mirandola, R., Prehofer, C., Wuttke, J., Andersson, J., Giese, H., Göschka, K.M.: On patterns for decentralized control in self-adaptive systems. In: de Lemos, R., Giese, H., Müller, H.A., Shaw, M. (eds.) Software Engineering for Self-Adaptive Systems II: International Seminar, Dagstuhl Castle, 24–29 Oct 2010 Revised Selected and Invited Papers, pp. 76–107. Springer, Berlin/Heidelberg (2013). https://doi.org/10.1007/978-3-642-35813-5_4
53. Weyns, D., Bencomo, N., Calinescu, R., Camara, J., Ghezzi, C., Grassi, V., Grunske, L., Inverardi, P., Jezequel, J.M., Malek, S., Mirandola, R., Mori, M., Tamburrelli, G.: Perpetual assurances for self-adaptive systems. In: de Lemos, R., Garlan, D., Ghezzi, C., Giese, H. (eds.) Software Engineering for Self-Adaptive Systems III: Assurances, Lecture Notes in Computer Science, vol. 9640. Springer, Cham (2017)
54. Wittenmark, B., Åström, K., Årzén, K.E.: Computer control: an overview. Technical report (2002)
55. Zhu, X., Wang, Z., Singhal, S.: Utility-driven workload management using nested control design. In: American Control Conference, Minneapolis, p. 6 (2006)
56. Zhu, X., Uysal, M., Wang, Z., Singhal, S., Merchant, A., Padala, P., Shin, K.: What does control theory bring to systems research? SIGOPS Oper. Syst. Rev. **43**(1), 62–69 (2009). https://doi.org/10.1145/1496909.1496922

Chapter 3
Challenges in Engineering Self-Adaptive Authorisation Infrastructures

Lionel Montrieux, Rogério de Lemos, and Chris Bailey

Abstract As organisations expand and interconnect, authorisation infrastructures become increasingly difficult to manage. Several solutions have been proposed, including self-adaptive authorisation, where the access control policies are dynamically adapted at run-time to respond to misuse and malicious behaviour. The ultimate goal of self-adaptive authorisation is to reduce human intervention, make authorisation infrastructures more responsive to malicious behaviour, and manage access control in a more cost-effective way. In this chapter, we scope and define the emerging area of self-adaptive authorisation by describing some of its developments, trends, and challenges. For that, we start by identifying key concepts related to access control and authorisation infrastructures and provide a brief introduction to self-adaptive software systems, which provides the foundation for investigating how self-adaptation can enable the enforcement of authorisation policies. The outcome of this study is the identification of several technical challenges related to self-adaptive authorisation, which are classified according to the different stages of a feedback control loop.

3.1 Introduction

A critical concern for organisations is the assurances that need to be provided regarding confidentiality, integrity, and availability of their computer-based resources. To provide such assurances, organisations use access control to protect

L. Montrieux (✉)
National Institute of Informatics, Tokyo, Japan
e-mail: lionel.montrieux@zalando.de

R. de Lemos
University of Kent, Canterbury, UK

University of Coimbra, Coimbra, Portugal
e-mail: R.Delemos@kent.ac.uk

C. Bailey
University of Kent, Canterbury, UK

Y. Yu et al. (eds.), *Engineering Adaptive Software Systems*,
https://doi.org/10.1007/978-981-13-2185-6_3

against unauthorised access. Regardless of adopting a fine-grained approach to access control, abuse of access is still possible. Any form of access, no matter how restrictive, presents the risk of attacks due to uncertainty in user behaviour. To accommodate for this risk, organisations employ a range of methods [32] to monitor and audit access within their systems and resources.

Traditionally, human administrators are relied upon to actively identify and drive changes in access control in response to detected abuse, natural organisational change, or identified errors in the criteria for access. It is challenging for human administrators to maintain a true awareness of the configuration of access, particularly within a run-time environment. With no complete view of access, obtaining assurances [21] against changes made to mitigate the abuse of access is limited. This potentially enables erroneous changes that cause a greater impact to the organisation over identified abuse. In addition, and as evident in case studies of historic insider attacks [11], the use of human administrators alone is inefficient in mitigating abuse in a timely manner. Improving on access control methodologies is one solution, yet such approaches [7, 25, 31, 42] are unable to actively mitigate abuse, since they are constrained to a static definition of the criteria for access control at run-time.

Implementations of authorisation infrastructures [14] must be capable in handling the dynamic aspect of risk at run-time, driven by the uncertainty in user behaviour. It is therefore necessary for such systems to actively observe how access rights are being used, in order to infer whether the current criteria and assignment of access are enabling a user to conduct malicious activity. A promising solution for the provision of dynamic support to authorisation infrastructures is the incorporation of self-adaptation.

Self-adaptive systems are systems that are able to modify their behaviour and/or structure in response changes that occur to the system itself, its environment, or even its goals [17]. Applying self-adaptive techniques to authorisation infrastructures enables the infrastructure to observe, reason, and act on its own configuration of access control. Through the use of a feedback loop [10], it is possible to employ a clear separation of concerns between the decision for access, and decision for a management change, therefore reducing the complexity in the criteria for access that dynamic access control approaches introduce.

The main contribution of this chapter is identification of several technical challenges associated with the self-adaptation of authorisation infrastructures. Another contribution is related to how the self-adaptation of authorisation infrastructures should be structured in order to handle parametric adaptations, i.e. the specification of access rights of subjects to resources, and structural adaptations, i.e. the enforcement of those specifications by controlling the subject's access to a resource.

The rest of chapter is organised as follows. Section 3.2 presents the concepts and terminology related to access control and authorisation infrastructures and provides a brief introduction to self-adaptive software systems. We review the related work on dynamic access control in context of self-protection in Sect. 3.3. Section 3.4 identifies, in terms of the key stages of a feedback control loop, like the MAPE-K loop, some challenges associated with the engineering self-adaptive authorisation infrastructures. Finally, Sect. 3.5 concludes the chapter and indicates directions for future work.

3.2 Background

The focus of this chapter is the application of self-adaptation in the management of privileges and the rendering of access control decisions. The goal is to reduce the need for human intervention while reducing cost and enabling systems to robustly adapt when responding to change. As such, the following section discusses some background topics: access control models, authorisation infrastructures, static and dynamic access control, self-adaptive authorisation infrastructures, and insider threats that we employ as a motivation for managing access control.

3.2.1 Access Control Models

In the literature, the terms authorisation and access control are sometimes used interchangeably. In this chapter, we define them as follows.

Definition 3.1 (Authorisation) Authorisation refers to the specification of whether a subject has access to a resource.

Definition 3.2 (Access control) Access control refers to the enforcement of authorisation by controlling (i.e. granting or denying) subject's access to a resource.

The goal of access control is to prevent unauthorised access to protected resources. A resource could be anything from a software system (e.g. web application and database) to an electronic device (e.g. electronic door lock and mobile phone). Through the specification of authorisation, captured in terms of policies, an organisation garners a certain level of protection from unwanted access.

Authorisation embodies two concepts: identities and permissions. An *identity* is a digital representation of a subject (a user), where a subject could be a human being, a system, or even a process [7]. An identity contains information about the subject, particularly relevant for authentication [43], where a subject must identify themselves, for example, entering in a username and password or use of biometrics [47]. Most importantly, an identity contains a set of the subject's access rights (also referred to as privileges [13]). Access rights, as the name suggests, represent a subject's right of access to a resource, used in accordance to a set of *permissions*. Once a subject obtains the required access right(s) to a resource, the subject is said to be authorised.

Access control models classify and define how permissions are expressed, who can define permissions, and what an access right looks like [37]. Arguably, the most adopted access control model in industry is the Role-based Access Control (RBAC) model [35], where recently 50% of the 150 companies surveyed by the National Institute of Standards and Technology (NIST) had adopted RBAC by 2010 [39]. RBAC introduced the notion of roles, whereby a role is assigned a set of permissions that enable access to a resource. Finally, the Attribute-based Access Control (ABAC) model [57] presents a more generic view of the RBAC model, where instead of roles, attributes (a type – value tuple) are used in order to collate and assign permissions.

3.2.2 Implementing Access Control Models

Traditionally, access control models have been implemented as bespoke components of information systems. A problem with this approach is the heterogeneous qualities of resources an organisation may wish to protect, often requiring each resource (e.g. a web application) to implement its own form of access control.

A solution to this problem is implementing access control models in a service-orientated way, as demonstrated by the eXtensible Access Control Markup Language [38]. XACML is a popular standard for implementing ABAC and RBAC models and provisions a reference architecture in which to guide implementation. XACML standardises the way in which identities and permissions are defined, communicated, and assessed in order for its reference architecture to render access control decisions. It does this through the use of authorisation policies (to express identities, privileges, and permissions as 'attributes' and 'rules') and the use of standardised protocols (e.g. SAML [37]). Authorisation policies embody the 'authorisation' aspect of an access control model, whereas the protocols support 'access control' via retrieval of privileges and deliverance of access control decisions.

XACML's reference architecture describes a set of components that exist to facilitate access to protected resources. The reference architecture defines a four-tier process to access control: *Enforce* requests and decisions to access, *Decide* upon access, *Support* retrieval of credentials and policies, and *Manage* administration of policies. This process is implemented through a set of conceptual components that when combined achieves access control (Table 3.1). These components are the enabling factors for controlling access, whereby in real systems that implement such components, access control can easily be monitored and managed. A key selling point of the XACML reference architecture is the separation between access control and resources, where access control primarily becomes a service that resources and users can rely upon.

Table 3.1 XACML components

Component	Description
Policy enforcement point (PEP)	Makes access requests and enforces access decisions
Policy decision point (PDP)	Evaluates access requests against policies to provide access decisions
Policy information point (PIP)	Contains subject identity information (attributes)
Policy retrieval point (PRP)	Contains ABAC authorisation policies to govern access decisions
Policy administration point (PAP)	The source of authority/system that issues access control policies

The XACML reference architecture has arguably sparked the rise of access control as a service, where we refer to such solutions as *authorisation infrastructures*.

3.2.3 Authorisation Infrastructures

An authorisation infrastructure [14] is a loose term for a collection of services and mechanisms that implement an access control model. There are a number of varying terms for authorisation infrastructures, such as the ones defined by authentication and authorisation infrastructures (AAIs) [30], XACML's reference authorisation architecture [38], and privilege management infrastructures [13]. We adopt the following rather simple definition for authorisation infrastructure.

Definition 3.3 (Authorisation Infrastructure) Authorisation infrastructures facilitate the management of identities, privileges and policies, and render access control decisions.

The key facet of authorisation infrastructures is the use of services that provide access control external to an organisation's resources. This implies a separation of duties between provisioning of services by the resources and the assessment of right to access [14, 30, 38]. Specifically, access control is implemented by the following key services:

Identity services responsible for the management of subject access rights, such as access rights and subject identifiers.

Authorisation services responsible for the evaluation of access rights against access control rules and decision of access.

The combination of both identity services and authorisation services should conform to an access control model (e.g. RBAC [35]). Based on existing implementations [14, 30, 38], identity services may authenticate subjects and maintain, assign, and release a subject's access rights (i.e. privileges) to authorisation services based on policies (e.g. Shibboleth's attribute release policy [33]). Examples of an identity service include directory services, such as the Lightweight Directory Access Protocol (LDAP) [28]. Other forms of identity services include credential issuing services (such as SimpleSAMLphp [49] and the Shibboleth identity provider [33]). These types of identity services not only maintain a subject's access rights (privileges) but can be configured to decide what access rights can be issued and released to given services across multiple domains. Authorisation services may validate and evaluate a subject's access rights against a set of policies (e.g. PERMIS's access control and credential validation policies [46]). Examples of authorisation services include the Axiomatics Policy Server [1], PERMIS stand-alone authorisation service [46] (both of which utilise the XACML standard to define access control policies), and the community authorisation service (CAS) [45].

Figure 3.1 defines a general model of an authorisation infrastructure that abstracts away from its varying implementations. With reference to the flow of communica-

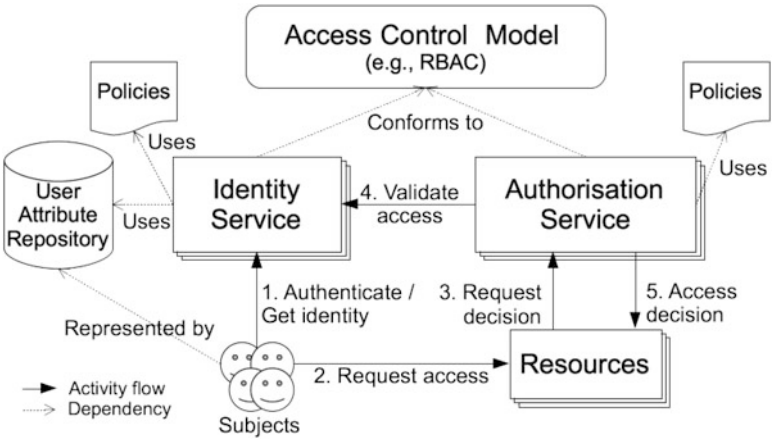

Fig. 3.1 General authorisation infrastructure model

tion to obtain authorisation, subjects (users) authenticate (1) with a given identity service that maintains a set of access rights for each subject. The authenticated subject can then request (2) access to a particular resource. The resource's policy enforcement point (PEP) communicates with an authorisation service (3), which can first validate (4) the subject's set of access rights and then decide upon access. The authorisation service sends a response back to the PEP with a message indicating whether authorisation should be granted or denied (5).

As already mentioned, authorisation infrastructures rely on policies to derive access control decisions. Authorisation infrastructures may utilise a variety of policy types, where an instance of a policy type will express rules relevant to a particular service within an authorisation infrastructure. For example, policies within authorisation services are used to define the constraints of access (i.e. RBAC role permission assignments), whereas policies and subject attribute repositories (e.g. LDAP [28]) within identity services contain or define what subjects have in terms of assigned access. With this in mind, there are four types of authorisation policies, which are defined as follows.

Access Control Policy An access control policy specifies the security controls of what credentials a subject must own in order to gain access to a set of protected resources, what obligations they must conform to, and what conditions they must meet.

Credential Validation Policy A credential validation policy defines what credentials an identity service is trusted to issue.

Delegation Issuing Policy A delegation issuing policy defines the trust in the extent of access a subject can delegate unto others.

Credential/Attribute Release Policy A credential/attribute release policy defines what information an identity service will release on behalf of a subject to any requesting authorisation services or resources.

Associated with policies and access rights is the notion of source of authority (SOA) and issuer [14]. A *source of authority* is the owner of a resource that establishes the rules of access (as policies) to their resources. An *issuer* is the identity service or person responsible for issuing to a subject a set of access rights, which are either stored in an attribute repository as unsigned or signed attributes [24] or are generated at time of request [37].

Lastly, an additional quality of authorisation infrastructures is the ability to operate in federated environments (i.e. components of an authorisation infrastructure become component systems managed across multiple organisations). This is often referred to as federated identity management, or federated access control [15, 22, 26, 53], and enables the sharing of access across multiple management domains (organisations). Various access control models are suitable for federated access control, demonstrated by several implementations [14, 33, 49].

3.2.4 Static and Dynamic Access Control

We have seen how access control is a key element when implementing authorisation infrastructures. However, one thing not addressed is the distinction between traditional (static) approaches to access control, to more recent (dynamic) advanced approaches. In a static approach to access control, a user's access rights are assessed against a set of security controls in order to determine access (e.g., RBAC [35]). This is limited since at time of access no additional context is assessed, such as the user's historical access, their location, time of day, or other external factors. With this in mind, static approaches are presumptuous in that, should a user have the necessary access rights, they should be awarded access.

Arguably, it is not always the case that access should be granted despite the user owning the necessary access rights. As such, there is a growing focus on dynamic approaches to access control that allow organisations to define a finer grain of control over access in response to varying risks, threats, and environment states. The definitions for static and dynamic access control are as follows.

Definition 3.4 (Static Access Control) Static access control refers to the evaluation of a subject's access rights against a set of immutable authorisation policies for deciding the subject's access to a resource, regardless of the context in which the request is made and evaluated.

Definition 3.5 (Dynamic Access Control) Dynamic access control refers to the evaluation of a subject's access rights against a set of authorisation policies for deciding the subject's access to a resource, taking into account the context in which the request is made and evaluated.

Dynamic access control differs from static access control because it is capable of employing various security controls that are related to changes in the state of the environment or protected resources and user activity. As such, an authorisation

policy may contain a diverse set of access control rules to accommodate a wide variety of scenarios (e.g. a rise in national security threat levels [31]). Appropriate access control rules are applied to requests for access in a mutually exclusive manner, given the context (i.e. state of the environment, such as user activity or time of day) that surrounds the request.

The overall goal of dynamic access control is to reduce human intervention, make access control more responsive to attacks, and more cost effective. Several techniques have been proposed, including resource usage [42], temporal properties [25], risk [31], and trust [8, 48]. For example, in usage control [42], a perception of user activity is maintained over time and evaluated against thresholds of usage (e.g. a staff member may not print more than 100 pages per day), alongside traditional access control rules (e.g. a user must be assigned the role of staff to print). Additionally, ABAC can be seen as a dynamic access control model given its ability to define permissions that can be valid for a multitude of system states.

3.2.5 Self-Adaptive Authorisation Infrastructures

With the goal of reducing human intervention, self-adaptation can be incorporated into existing authorisation infrastructures, thus enabling these infrastructures to manage themselves, at run-time, the definition of authorisation policies and process of access control. In particular, the focus of this chapter is how self-adaptation can be integrated with authorisation infrastructures and how authorisation infrastructures can self-protect against insider threats.

3.2.5.1 Self-Adaptation

Self-adaptation enables a system to adjust itself in response to changes that might affect itself or its environment. Self-adaptive systems can be defined as follows.

Definition 3.6 (Self-Adaptive Systems [17]) 'Systems that are able to modify their behaviour and/or structure in response to changes that occur to the system itself, its environment, or even its goals'.

Although there are several reference models for self-adaptive systems [27, 29, 41], most of them share the common use of a feedback loop [10, 18, 20]. In this chapter, we adopt as a feedback control loop, the Monitor, Analyse, Plan, Execute – Knowledge (MAPE-K) reference model [27], as shown in Fig. 3.2. In this figure, the Controller, which embodies the stages of the MAPE-K reference model, observes (via probes) and adapts (via effectors) a target system – represented as red arrows. The *Monitor* stage obtains the state of the target system and its environment. The *Analyse* stage analyses the state of the target system and its environment in order, first, to decide whether adaptation should be triggered (*solution domain*) and, second, to identify the appropriate courses of action in case adaptation is required

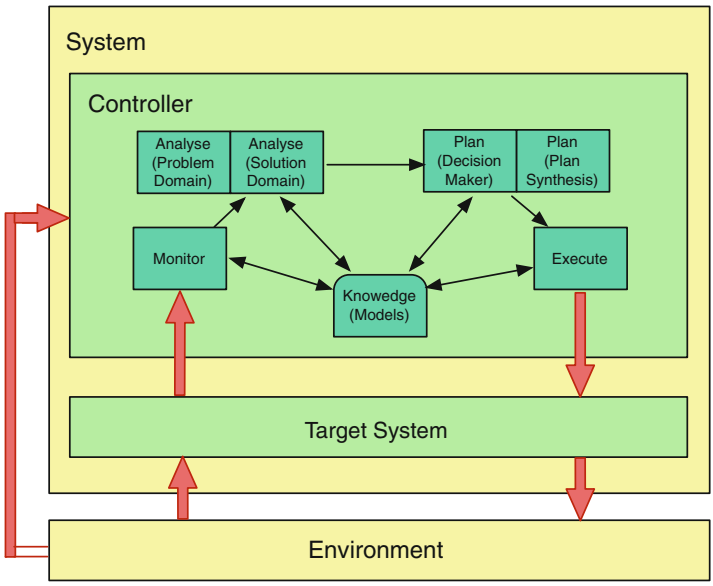

Fig. 3.2 Modified version of the MAPE-K reference model for autonomic computing [27]

(*problem domain*). The *Plan* stage, first, selects amongst alternative course of action those that are the most appropriate (*decision-maker*) and, second, generates the plans that will realise the selected course of action (*plan synthesis*). The *Execute* stage executes the plans that deploy the course of action for adapting the system. Finally, *Knowledge* represents any information related to the perceived state of the target system and environment that enables the provision of self-adaptation, which is shared by all the stages.

Applying the MAPE-K reference model, we view an authorisation infrastructure as the target system, and all the rest, including the users and protected resources, as the environment. The role of a controller[1] seeks to monitor both the target system and the environment in which to drive changes at run-time within the authorisation infrastructure. With this in mind, self-adaptation is capable of extending traditional approaches to access control, where such approaches become capable to respond to unplanned states, evolve to changing user needs, and maintain assurances in confidentiality, integrity, and availability of resources.

[1] Also referred to as the self-adaptive layer.

3.2.5.2 Self-Adaptive Authorisation and Self-Adaptive Access Control

Self-adaptive authorisation has already been proposed by Bailey et al [2–5], where legacy-based authorisation infrastructures have been shown to mitigate, at run-time, attacks via the adaptation of authorisation (e.g. adaptation of authorisation policies and subject privileges). We define self-adaptive authorisation as follows.

Definition 3.7 (Self-Adaptive Authorisation) Self-adaptive authorisation refers to the run-time adaptation of the specification of whether a subject has access to resources.

The incorporation of self-adaptation into authorisation has highlighted a number of challenges that this chapter aims to address, including the engineering of self-adaptive authorisation infrastructures and practicalities of operating such systems at run-time. First, it is important to identify the differences between static approaches to access control (i.e. traditional, such as RBAC), dynamic approaches (i.e. adaptive, such as risk based), and self-adaptive ones.

Let us consider a subject requesting access to a resource outside of normal working hours, who then abuses such access in order to jeopardise the confidentiality of a resource. A static approach (i.e. static access control) will evaluate access based on purely the subject's access rights alone, without considering the time of day, or the subject's activity. A dynamic approach (i.e. dynamic access control) may select, from a pre-existing set of access control rules, a rule applicable for that time of day, using environmental attributes and the subject's access rights. On the other hand, a self-adaptive approach may, at run-time, generate, modify, or remove the active set of access control rules (e.g. deploying a new authorisation policy or revoking a set of user access rights) should a user be detected while abusing their access rights outside of normal working hours. Additionally, modifications instructed by a self-adaptive approach are based on a maintained perception of state of its target system and its environment.

Self-adaptive authorisation alone has some limitations. Specifically, it is limited to only mitigating attacks (e.g. insider threats) within the boundaries of an authorisation infrastructure's implemented access control model, where adaptation is primarily parametric. Should services of an authorisation infrastructure suffer an attack, or the implemented access control model becomes vulnerable, an additional scope of adaptation is needed. As such, it is important to address the possibility of self-adaptive access control, which we define as follows.

Definition 3.8 (Self-Adaptive Access Control) Self-adaptive access control refers to the run-time adaptation of the enforcement of authorisation by controlling the subject's access to a resource.

Figure 3.3 emphasises the association between the self-adaptive authorisation and self-adaptive access control, which allow us to mitigate attacks more effectively and efficiently, depending on the type of attack observed. Authorisation is the collection of policies that govern access, while access control is the process in how an access decision is achieved. From the diagram, we can see a distributed

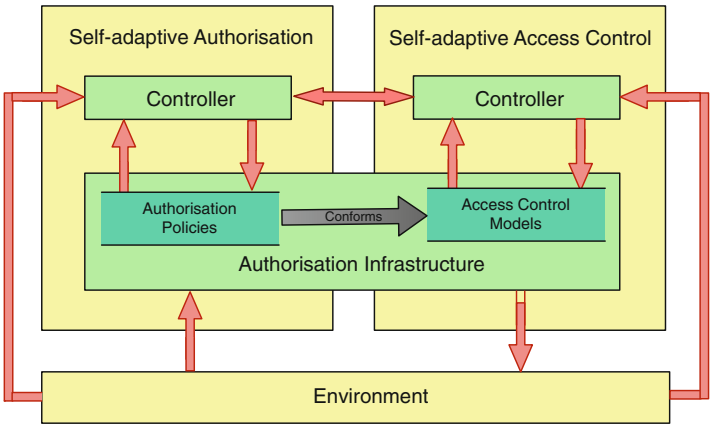

Fig. 3.3 Self-adaptive authorisation and self-adaptive access control

control topology of two controllers operating together in mitigating potential attacks originating from the environment of the authorisation infrastructure. The controller associated with self-adaptive authorisation observes activity of the authorisation infrastructure and its environment in order to gain a perception of malicious behaviour with relevance to the current state of authorisation policies. Should malicious behaviour be observed, this controller can adjust deployed authorisation policies to mitigate attacks. Similarly, the controller associated with self-adaptive access control may observe the authorisation infrastructure and its environment in order to identify if the current state of the employed access control model is fit for purpose. For example, external threats may warrant additional steps in validating subject credentials, and as such, the controller may deploy credential validation services [14] between policy decision and policy enforcement points [38]. Based on the above, we define self-adaptive authorisation infrastructures as follows.

Definition 3.9 (Self-Adaptive Authorisation Infrastructure) Self-adaptive authorisation infrastructures refer to the run-time adaptation of the collection of authorisation policies and their enforcement.

3.2.5.3 Self-Protection

Self-protection is of particular relevance since the goal of this work is to manage access control in order to mitigate abuse of access. Self-protecting systems can be defined as follows.

Definition 3.10 (Self-Protecting System [59]) Self-protecting systems are a class of autonomic systems capable of detecting and mitigating security threats at run-time.

There are various self-protective solutions that seek to detect and mitigate malicious behaviour. However, few works exist that are able to concretely address self-protection with a view to mitigate the abuse of access. While many systems appear to be self-protective, such as intrusion response systems [34, 51, 52], many are only adaptive and lack an awareness of 'self'. A self-protecting system is clearly demonstrated by Yuan et al.'s architectural-based self-protection framework [58], where a system maintains a modelled state of its own system architecture in which to guide mitigation of threats.

3.2.6 Insider Threats

Insider threat refers to an organisation's risk of attack by their own users or employees. It is fast becoming a prominent topic that organisations need to address, as highlighted by recent scandals in the media [6, 9, 54]. This is particularly relevant to access control, where the active management of authorisation has the potential to mitigate and prevent users from abusing their own access rights to carry out attacks. The CERT Guide to Insider Threats (Cappelli et al.) [11] defines malicious insider threats as the following.

Definition 3.11 (Insider Threat [11]) 'A malicious insider threat is a current or former employee, contractor, or business partner who has or had authorised access to an organisation's network, system, or data and intentionally exceeded or misused that access in a manner that negatively affected the confidentiality, integrity, or availability of the organisation's information or information systems'.

Capelli et al. [11] classify three types of insider threat: *sabotage*, where malicious users attempt to damage or corrupt organisational resources; *theft* of intellectual property, where organisational resources are stolen and distributed; and *fraud*, where activity is covered up or information is used to commit crimes, such as falsifying money transfers.

A common characteristic of insider threat is that malicious insiders utilise their knowledge of their organisation's systems, and their assigned access rights, to conduct attacks. This places a malicious insider in a fortuitous position, whereby the insider (as an authorised user) can cause far greater damage than an external attacker, simply due to their access rights [12]. Such form of attack is representative of the attacks that many organisations consider to be most vulnerable from, being the abuse of privileged access rights by the employees of an organisation [40].

Unless additional measures are put into place, malicious insiders can abuse existing security measures, where current approaches fail to robustly adapt and respond to the unpredictable nature of users. For example, traditional approaches to access control assume that if a user has authenticated, and has the required access rights, access to resources should be given. While there are a number of novel techniques that enable the detection of insider threat [23, 36, 50], there is little research that utilises such techniques within an automated setting. Many existing approaches require analysis by human agents to identify and execute resultant actions to mitigating attacks.

3.3 Related Work

In this section, we review some related work on self-protection in the context of detection and mitigation of insider threat.

Self-protecting systems are a specialisation of self-adaptive systems with a goal to mitigating malicious behaviour. In the following, we discuss the few works that have demonstrated self-protection within the context of mitigating insider attacks. In particular, we discuss two self-protection approaches based on the state of access control and one approach based on the state system architecture.

One of the approaches to self-protection via access control is SecuriTAS [44]. SecuriTAS is a tool that enables dynamic decisions in awarding access, which is based on a perceived state of the system and its environment. SecuriTAS is similar to dynamic access control approaches, such as RADac [31], in that it has a notion of risk (threat) to resources and changes in threat leads to a change in access control decisions. However, it furthers the concepts in RADac to include the notion of utility, whereby given a perceived state of the system and its environment, the optimum set of security controls are used. This is achieved through an autonomic controller that updates and analyses a set of models (that define system objectives and vulnerabilities, threats to the system, and importance of resources in terms of a cost value) at run-time. The autonomic controller deploys optimal security controls (i.e. access control constraints) within the system, changing the conditions of access. A novel aspect of this work is that it is aimed towards physical security, whereby a resource (e.g. a computer terminal or handheld device) is stored within an office (also considered to be a resource), for example. SecuriTAS may change the conditions of access to the office based on the presence of high-cost resources or the presence of highly authorised staff.

Another form of self-protection in access control is positioned by SAAF [4], a Self-Adaptive Authorisation Framework. SAAF's goal is to make existing authorisation infrastructures self-adaptable, where an organisation can benefit from the properties of dynamic access control without the need to adopt new access control models. This is achieved through a globally centralised autonomic controller that monitors the distributed services of an authorisation infrastructure to build a modelled state of access at run-time (i.e. deployed access control rules, assigned subject privileges, and protected resources). Malicious user behaviour observed by a SAAF controller is mitigated through the generation and deployment of authorisation policies at run-time, preventing any identified abuse from continuing. Adaptation at the model layer enables assurances and verification that abuse can no longer continue. In addition, model transformation has been shown to generate authorisation policies from an abstract model of access. This has the potential to enable the generation of policies specific to many different implementations of access control.

The main difference between SecuriTAS and SAAF is that SecuriTAS positions its own bespoke access control model and authorisation infrastructure that incorporates self-adaptation by design. SAAF, on the other hand, is a framework

that describes how existing access control models and authorisation infrastructures can be made self-adaptive and, as such, configured to actively mitigate insider threat. With that said, both approaches demonstrate an authorisation infrastructure's robustness in mitigating insider attacks, by ensuring that authorisation remains relevant to system and environment states (and preventing continuation of attacks by adaptation of security controls).

In contrast to self-protection via access control, architectural-based self-protection (ABSP) [58] presents a general solution to detection and mitigation of security threats, via run-time structural adaptation. Rather than reason at the contextual layer of 'access control', ABSP utilises an architectural model of the running system to identify the extent of impact of identified attacks. Once attacks or security threats have been assessed, a self-adaptive architectural manager (Rainbow [19]) is used to perform adaptations to mitigate the attack. One adaptation example the approach offers is to throttle network connections to a server, in order to disrupt ongoing attacks. Another example is the deployment of *application guards* where a protective wrapper is deployed around architectural components (e.g. a web server). These provide mitigation measures that improve upon the integrity of architectural components (i.e. the encryption of session ids susceptible to hijacking). ABSP shares a number of similarities with intrusion response and prevention systems, particularly with the scope of adaptations that ABSP can perform (e.g. structural adaptation against network devices and connections). However, because ABSP maintains a notion of 'self', it is able to reason about the impact of adaptations and provide assurance over adaptation before adapting its target system.

3.4 Challenges in Engineering Self-Adaptive Authorisation Infrastructures

In this section, we identify some challenges for engineering self-adaptive authorisation infrastructures in the context of the MAPE-K loop. For each of the stages of the MAPE-K loop, we discuss what are the challenges specifically associated with self-adaptive authorisation infrastructures, looking, in particular, into issues related to insider threats. For example, what types of probes are needed for the Monitor stage, how to generate dynamic plans in the Plan stage, and how to perform policy updates in Execution, etc. For each of the challenges, we identify and describe the challenge and discuss their relevance in the context of authorisation infrastructures regarding insider threats.

3.4.1 Monitor

The size of what needs to be monitored and the ability of the monitoring to adapt its own probes and gauges are the two dimensions that will influence the complexity of the Monitor stage. Since self-adaptive authorisation infrastructures have no control over their environment, it is impossible to foresee *all* the environment changes that might affect the system. Some changes can easily be detected by the probes and gauges of the self-adaptive authorisation infrastructure, while some others can remain oblivious if the appropriate probes and gauges are not provided. In order to avoid the risk of the infrastructure missing important information, it is necessary to dynamically adapt (1) what needs to be monitored and (2) the type of probes and gauges required.

3.4.1.1 Active Monitoring

With passive monitoring, static probes and gauges are set up at deployment time, to monitor the authorisation infrastructure and its environment. The probes and gauges are static since they cannot be redeployed or removed at run-time, nor can they be reconfigured.

While it may be tempting to monitor a wide range of environment resources, monitoring comes at a cost. It has an impact on performance, and may affect other requirements, such as the users' privacy. Within a changing self-adaptive authorisation infrastructure and its environment, the right balance between data collection and performance or privacy is likely to evolve.

Challenge The challenge is the provision of active (or proactive) monitoring for reducing the amount of traffic related to monitoring considering that some of the analysis can be performed by the probes and gauges themselves. Moreover, proactive monitoring requires the availability of smart probes and gauges, able to adapt to what they monitor.

Relevance The key motivation for proactive monitoring in self-adaptive authorisation infrastructures is to make dynamic access control more resilient to changes, thereby allowing the infrastructure to better detect and react to insider threats. The detection of insider threats relies on monitoring a wide range of resources from the environment of the authorisation service with the purpose of profiling the status and activity of subjects inside the organisation. As the monitoring might be outside the ownership of the authorisation service, special probes need to be synthesised and deployed that might be constrained by privacy issues, for instance.

3.4.1.2 Run-Time Synthesis of Probes and Gauges

The synthesis of probes and gauges at run-time is one way of achieving active monitoring. The decision to synthesise probes and gauges should not be restricted to the Monitor stage of the MAPE-K loop; other stages may well require further data regarding the system or its environment. The synthesis, configuration, and deployment of probes and gauges should be the responsibility of the Monitoring stage, but it should be supported by different control loop.

Challenge The challenge is the ability to synthesise probes and gauges at run-time, in response to new or emerging attacks. These probes and gauges, once deployed, should improve the resilience of the self-adaptive authorisation infrastructure against unexpected changes.

Relevance The run-time synthesis and deployment of probes and gauges in self-adaptive authorisation infrastructures can help to cope with the unpredictable nature of an attack. There are no guarantees that what is being monitored is sufficient to identify a whole range of attacks, hence the need to autonomously synthesise and deploy a probe or a gauge that would be able to examine novel system attributes.

3.4.1.3 Mutating Gauges

Mutating gauges are gauges that are able to change themselves, either randomly or guided, in order to identify unknown behavioural patterns that might be related to an attack. If the monitoring system needs to have the capability to detect autonomously previously unknown patterns of attack, one way to enable this is to generate new detectors by mutating existing ones.

Challenge The challenge is to generate and deploy these mutating gauges for examining real-time or past data to identify unexpected interactions that an authorised subject might have with the system being protected. These mutating gauges can be used to provide additional evidence, with some degree of confidence, that an attack is, or has been, taking place.

Relevance Since the environment of authorisation infrastructures is dynamic and unpredictable, one should not expect to know about all possible attacks before deploying the system. The ability to deploy mutating gauges would enable the detection of new forms of attack by simply looking for unknown anomalies, and this would be enabled by the random nature of these gauges, i.e. there is no implicit expectation of what they should be able to detect. Let's consider the case in which a gauge monitors the access to a service by authorised users. A possible change in the environment of this service is the deployment of a new version of the server providing the service, and this might result in changing the format of the logs that the gauge is supposed to monitor. Either the original gauge becomes ineffective, or it needs to be manually reconfigured. Alternatively, once a change is detected in the log format, a mutating gauge may be able to automatically adapt itself in order to

understand the new format. Another possible usage of mutating gauges would be to enable the perpetual analysis of logs in order to identify attacks. During run-time, as an offline activity, different gauges could be dynamically generated by mutation, and these would analyse the logs for identifying attacks previously unknown.

3.4.1.4 Incomplete Information

Incomplete information refers to the situation in which the Monitor stage is not able to provide all the information needed by the other stages of the control loop. This might be due to limited monitoring capabilities, and because of that, the monitoring stage has to find alternative ways of obtaining the missing information.

Challenge Identify and select what to monitor in order to compensate the missing information, and know where it is safe to make assumptions about the unknown.

Relevance Monitoring has a cost, especially when considering insider threats. The detection of insider threats relies mostly on data from the environment, and since the environment of an authorisation infrastructure is broad and fluid, in the sense that it is difficult to establish its clear boundaries, this has an effect on the data that is collected. Therefore, the system will likely have to deal with incomplete information, which in the Analyse stage might lead to more false positives regarding insider threats. One way to compensate for incomplete information is for the gauges themselves to provide a level of confidence regarding the information that is forwarded to other stages of the control loop.

3.4.1.5 Automatic Feature Identification

During system operation certain probes may cease to function, either maliciously or accidentally. In order not to lose the features being monitored through that probes, the system should be able to recover some or all of those features by making use of the information provided by other probes. The assumption is that several features can be associated with a probe and that these features can be extracted and combined with other features from other probes in order to reconstruct totally or partially the information lost from an unavailable probe.

Challenge The challenge is for the system to be able to automatically extract features from its probes, and recombine those features as necessary, thus exploiting some intrinsic redundancy that may exist amongst the probes. At run-time, this should be achieved by combining and reconfiguring features that are associated with the information provided by several probes.

Relevance This challenge is relevant to self-adaptive authorisation infrastructures because it helps to increase the system's resilience against run-time threats to probes and gauges, whether they are intentional or accidental.

3.4.2 Analyse

The Analyse stage is made of two consecutive parts: the problem domain analysis and the solution domain analysis. The problem domain analyses the data provided by the Monitor stage in order to identify changes that the system may have to respond to. The solution domain analysis occurs after a problem has been identified and is concerned with generating possible alternative solutions that are able to handle the problem. The problem domain analysis can be further divided in two parts: the identification of potential problems and the assessment of the identified problem in order to prioritise the mitigation. In some cases, a problem may be identified as sufficiently serious to be addressed immediately. At the other end of the spectrum, some problems may be acknowledged, but ignored, as they are deemed not critical enough to cause adaptation.

In the following, first, we present the challenges related to the problem domain: anomaly detection, signature-based detection, case-based detection, diagnosis, and normality detection. Then, we present those challenges related to both problem and solution domains by clearly identifying how these are related to each of the domains: perpetual evaluation, threat management, and risk analysis.

3.4.2.1 Anomaly Detection

Anomaly detection is related to the ability of the controller to identify any behaviour that deviates from what is perceived to be acceptable. Since we are essentially dealing with sociotechnical systems for which it is almost impossible to establish, from the outset, all their possible behaviours, it is extremely challenging to clearly distinguish normal from abnormal behaviour, i.e. what is acceptable and what it is not. First, there is the uncertainty of the context of the system that might influence whether a particular behaviour is deemed to be normal or abnormal. Second, there are the previously unknown or unexpected behaviours that need to be classified according to profiles of similar class of behaviours.

Since there is no single technique that should be able to accurately detect a wide range of anomalies, one way of reducing the number of misclassifications is to use diverse techniques whose outcome should be fused for providing confidence in the classification. In the following, after introducing anomaly detection challenge, we present, as an example, two specific complementarity anomaly detection techniques that can be used for improving both the responsiveness and coverage when detecting anomalies.

Challenge The key challenge in anomaly detection is to be accurate when detecting anomalies under uncertainty in order to reduce misclassifications, specifically in the context of insider threats. Since misclassifications cannot be eliminated, it is important to associate with those classifications levels of uncertainty.

Relevance The ability of detecting anomalies should precede the system capability of handling insider threats. Since it is difficult to accurately identify an attack, uncertainty levels should be considered, so the system can evaluate a particular detection against its context. The objective is to reduce the number of false positives and false negatives that might have detrimental consequences upon self-adaptive authorisation infrastructures.

3.4.2.2 Signature-Based Detection

Signature-based detection is a special case of anomaly detection (see Sect. 3.4.2.1), where domain analysis is performed by matching the data provided by the Monitor stage against signatures of known problems. A signature is a pattern that should be matched against the data provided by the Monitor stage, such as an IP address, a particular regular expression in a log file, a URL, a version of some software, etc. Signature-based detection may require the matching of several individual signatures to identify a threat. They are relatively easy to automate. Since signatures refer to precise pieces of information, it is possible to completely automate their recognition and therefore the identification of threats. With a sufficiently expressive language to write the signatures and their interactions, complex analysis can be performed to discover advanced threats. Administrators should also be allowed to define signatures, as well as combinations of signatures, and associate them with threats.

Challenge Since the signature-based detection is a static technique, the challenge is to be able to synthesise new signature-based detectors during run-time.

Relevance Signature-based detection is best suited to detect threats that are known and well understood in advance. However, in the context of self-adaptive authorisation infrastructure, the efficacy of static signature-based detectors is quite restrictive considering that both the attacks and the infrastructure can change. Thus, there is the need for the self-adaptive authorisation infrastructure to be able to generate dynamically new signature-based detectors that are able to detect unknown threats efficiently at run-time.

3.4.2.3 Case-Based Detection

Case-based detection is another special case of anomaly detection, where the focus is on observing subjects' behaviours, which are harder to model and, hence, harder to automate. Where signature-based detection attempts to identify well-defined actions performed by malicious subjects, case-based detection observes the malicious behaviour of subject and allows for decisions to be made based on the subject's behaviour model. Moreover, instead of absolute thresholds for identifying anomaly detection, relative thresholds comparing users behaviours can be used.

Challenge The challenge in case-based detection involves recognising a behaviour that may not be explicitly forbidden but still suspicious.

Relevance It may be the case that a subject will try to circumvent signature-based detection since signature-based detection works by using thresholds and precise patterns of attacks. This is where case-based detection becomes useful. The attacker may be slowed down because of their efforts to avoid detection, but that does not mean that the threat does not need to be addressed. Case-based detection is a good way to complement signature-based detection because of its ability to detect and act upon those types of threats, although it is more difficult to be fully automated.

3.4.2.4 Diagnosis

When an attack is detected, the system may try to identify the source of the attack, how it was performed, what damage it caused or is causing, and which vulnerability was used to carry it out. Diagnosing an attack allows the system to better understand it and therefore to make better decisions to defend against it.

Challenge The challenge of diagnosing self-adaptive authorisation infrastructures is the ever-changing type of attack and the new vulnerabilities that might be introduced during adaptation.

Relevance Identifying the source of the attack and the vulnerability exploited is key to stopping it to propagate, as well as making sure that it does not happen again. A self-adaptive authorisation infrastructure that can understand where attacks come from and how they are carried out will be more resilient than a system that can only identify them without understanding what caused them to be successful.

3.4.2.5 Resuming Normality

When an attack is over and a threat does not anymore pose danger, or when a particular risk that had previously identified has been mitigated, the system should be able to undo the restrictive measures that were taken for protecting the system against the attack or the likelihood of an attack. This would be more relevant if the restrictive measures taken affected the system's normal operation. This should be done without exposing the system to other attacks.

Challenge After taking measures to protect the system against attacks, the challenge is when to undo some of the restrictive measures and what measures should be put in place in order maintain a balance between usability and security.

Relevance Measures taken to prevent or mitigate attacks, in the context of self-adaptive authorisation infrastructures, often take the form of reduced capabilities for users or more stringent authorisation procedures. If the system were not able to scale back some of the measures taken after the event that triggered them has occurred,

then the system would tend towards locking all the users out of the system. It is therefore crucial that the system is able to always strike the correct balance between usability and security.

3.4.2.6 Perpetual Evaluation

When the controller is not adapting the target system, it can run background tasks to enhance the resilience of the self-adaptive authorisation infrastructure. Perpetual evaluation is one such task, which stands for the continuous analysis of either the problem or solution domains.

Challenge The challenge associated with the perpetual evaluation of the problem domain is the identification of vulnerabilities and attacks that might affect the self-adaptive authorisation infrastructure. On the other hand, the challenge associated with the perpetual evaluation of the solution domain is the provision of assurances regarding the quality of services provided by the self-adaptive authorisation infrastructure.

Relevance Perpetual evaluation can be used alongside traditional evaluation in order to improve the coverage in detecting insider attacks, localise vulnerabilities, and enhance the provision of assurances. This can be done either proactively or reactively.

If insider attacks can be predicted to occur depending on some observable pattern of behaviour, adaptation can be proactive, and the same applies to evaluation. Since the proactive perpetual evaluation does not block any immediate adaptation, it can only inform future adaptations. While traditional evaluation can make fast, but imperfect, decisions, the reactive perpetual evaluation compliments traditional evaluation by confirming that the adaptation satisfies the system goal, or point to issues that may require a rollback, or further adaptation. This is possible because the reactive perpetual evaluation can afford to take longer to complete and consider more data or more stringent constraints. This is especially useful in scenarios where timeliness of adaptation is important, such as the response to insider threats.

3.4.2.7 Threat Management

There may be several simultaneous attacks detected or vulnerabilities identified, and responses to these in the form of adaptations should be prioritised. Furthermore, responses may increase the attack surface or weaken other security measures.

Challenge In the problem domain, the ability to prioritise attacks and vulnerabilities is a challenge associated with threat management, which should take into account the threats' potential impact on the system's operations, and attempt to take preventive measures, to ensure that future threats can be addressed.

In the solution domain, the challenge associated with threat management is the ability to rank alternative responses and to ensure that a response does not increase the system's attack surface or weakens its security measures.

Relevance Any perceived attack or vulnerability should not be considered in isolation from its current or historical contexts; otherwise problem domain analysis might be incomplete, thus producing outcomes that might undermine the mitigation of threats. Likewise, from the solution domain perspective, any measure to handle the perceived attack or vulnerability should take into account other measures either being processed or already processed. The goal is to reduce the amount of resources needed for handling the attack or the vulnerability and minimise the risk of introducing new vulnerabilities. Moreover, considering that known vulnerabilities might exist in the authorisation infrastructure, these should be taken into account when analysing measures for mitigating a perceived attack.

3.4.2.8 Risk Analysis

When perceived to be under attack, an authorisation infrastructure can use risk levels to rank alternative responses and select the most appropriate one. Factors that can influence the risk level include the coverage of the evaluation, the severity of the attack and/or the vulnerability, and the impact of countermeasures on the system's operations.

Challenge The challenge of risk analysis in the problem domain is to determine the seriousness of an attack, which should establish the appropriate response level. Regarding the solution domain, risk analysis should guide the selection of the most appropriate response when several options are available.

Relevance In the problem domain, depending on the perceived risk, attacks and vulnerabilities may need to be dealt with immediately, while others may allow for a delayed response, or no response at all, at little to no cost on the system's security. Whether a self-adaptive authorisation infrastructure shall react to an attack should depend on the risk associated with the attack and/or vulnerability: the probability of an attack to be successful, and the impact the attack might have on the system in case it is not mitigated.

Regarding the solution domain, adaptation may be expensive, whether in terms of time, computation resources, or inconvenience to legitimate users through degradation of the service. Therefore, a sophisticated authorisation infrastructure could use risk analysis to prioritise the order in which attacks should be addressed and when and how to deal with them.

3.4.3 Plan

The Plan stage is made of two consecutive parts: decision-making and plan synthesis. The purpose of decision-making is to select the most appropriate solution amongst the alternatives provided by the solution domain analysis. Below, we have identified three challenges associated with decision-making: decision-making in a federated authorisation infrastructure, randomising decisions, and denial of service. The goal of plan synthesis is to generate a plan that implements the selected solution. We identified six challenges related to the plan synthesis: robust plans, controller capabilities, and infrastructure boundary.

3.4.3.1 Decision-Making in a Federated Authorisation Infrastructure

There are several benefits associated with federated authorisation infrastructures, being one of them the ability of authenticating users using third parties. However, these pose additional challenges to the planning phase, compare with a simpler, centralised infrastructure over which a single entity or user has complete control. The selection of the best solution amongst alternatives, identified during the analysis problem domain, needs to consider the self-interests of the different parties of the federation. The component systems of a federated authorisation infrastructure may have conflicting interests and goals and varying constraints (e.g. an identity provider service may conflict with a service provider). Yet it is important to be able to select the best solution amongst those identified in the analysis solution domain while satisfying the goals and constraints of all the components in the federated authorisation infrastructure.

Challenge Decision-making in a federated authorisation infrastructure should take into account the potentially conflicting goals of all the parties in the infrastructure and negotiate a solution that satisfies them all. This may require a solution that is not optimal, but 'good enough', and acceptable to all parties involved.

Relevance In a self-adaptive federated authorisation infrastructure, it is expected for third parties to undergo some kind of change, for example, involving their goals or their deployment. This should have an impact on how the different parties collaborate in order to maximise each party self-interest. However, adaptation decisions that involve federated authorisation infrastructures may require negotiation between several stakeholders. If all the component systems' goals cannot be satisfied, then a self-adaptive authorisation infrastructure may have to consider stopping its collaboration with some or all of the component systems with whom a compromise could not be reached.

3.4.3.2 Randomising Decisions

If the decision-maker, for particular operational context, always selects the same strategy, then a new vulnerability is being introduced. If an attacker, while interacting with the system or observing its behaviour, is able to establish deterministically the response of the self-adaptive authorisation infrastructure, the attacker may be able to take advantage of this adaptation and cause harm to the system.

Challenge The selection of an adaptation solution amongst several more or less equally acceptable options should be randomised, in order to prevent an attacker from learning about the system's response to a particular output, thus reducing the attack surface.

Relevance Self-adaptive authorisation infrastructures could be targeted by attackers wishing to exploit a new vulnerability introduced by the controller. The nature of the adaptation measures that can be taken poses at least two threats. First, the attackers could trick the self-adaptive system into banning users, or groups of users, even if only for a limited amount of time, causing disruptions in the users' ability to use the service. Second, the attackers could trigger a denial of service attack by forcing very frequent changes in the authorisation policies, which would overwhelm the system.

3.4.3.3 Denial of Service

Denial of service (DoS) aims to make resource unavailable to their legitimate users, for example, by flooding a server with bogus requests that waste computing resources. An attacker could use the self-adaptation mechanism for this purpose, preventing legitimate users from using the service.

Challenge As a challenge, the self-adaptive authorisation infrastructure should be able to analyse the triggers for self-adaptation, identify their source and frequency, and react accordingly in order to avoid the system to become unusable.

Relevance Authorisation infrastructures for which the execution of self-adaptation requires reloading configuration files, restarting services, or interrupting or cancelling long-lived operations are particularly vulnerable to DoS attacks. If the attacker finds a way to trigger self-adaptation often enough, the system may become unusable for legitimate users. In this case, the self-adaptation mechanism itself is the attack vector used by the attacker to perpetrate his attack. If the system detects a possible DoS attack, it may then switch to a less obtrusive means of self-adaptation if available or disable self-adaptation for some time. Another option would be to cap the number of self-adaptation operations that disturb the service for a specified time period.

3.4.3.4 Robust Plans

Some of the activities in a plan may be more likely to fail than others. This could be related to complex interactions between components of a federated authorisation infrastructure. This can be caused by software or hardware failures or simply because assumptions made during the conceptualisation of the plan cease to be true during its execution.

Challenge In a self-adaptive authorisation infrastructure, the challenge is to obtain a robust plan that should be able to handle failures in one or several of its activities while minimising service interruptions. The plan should incorporate redundancy in its activities, or the ability to rollback to a previous working secure state, in case an activity fails.

Relevance Authorisation infrastructures involve various component systems, as well as a number of policies whose interactions determine who gets access to what. If a plan is not robust enough, the infrastructure could be left in an intermediate insecure and unstable state, i.e. some authorisation decisions could allow unauthorised subjects access to sensitive data. If a plan is rolled back, then the infrastructure is again vulnerable to the insider threat that had triggered the (aborted) adaptation. None of these scenarios are acceptable, and, therefore, the plan should be as robust as possible in order to deal with any unexpected issue arising during its execution. This can be achieved by enabling the controller to generate abstract plans (i.e. a plan that does not depend on any particular implementation, it can support several alternative implementations) that can be instantiated into concrete plans during their execution. In case a particular instantiation of an activity fails during its execution, the abstract plan should incorporate enough redundancy in order to activate an alternative instantiation.

3.4.3.5 Controller Capabilities

What a controller is able to achieve in a self-adaptive authorisation infrastructure is restricted by its capabilities. These capabilities are related to what the controller is able to observe and control and its computational and algorithm resources. Limitations on the controller's capabilities might have an impact on the plans that a controller is able to synthesise, and these limitations should be incorporated into plans.

Challenge In a self-adaptive authorisation infrastructure, because of its nature, it is difficult to forecast changes that might affect the system and its environment; the challenge is for the controller to be able to identify its own limitations. In case an operational boundary is reached, the controller should be able to act, either by shutting itself down or invoke another alternative controller, for example.

Relevance While synthesising a plan, there is a risk that some implementations are not able to support activities of the plan. For example, the controller of a self-

adaptive authorisation infrastructure is able to synthesise plans that only contain activities that rely on XACML implementation of Policy Enforcement Point (PEP), while the actual components of the infrastructure rely on other implementations rather than XACML.

3.4.3.6 Infrastructure Boundary

In a federated authorisation infrastructure, some components can be managed by third parties, and this should be captured by control boundaries that can be dynamic according to the role of the components of the infrastructure. These components may have different or even conflicting goals; hence, negotiations between components are needed in order to maximise their self-interest. In a self-adaptive authorisation infrastructure, the controller may only have partial control, or no control at all, over some of the components of the federated authorisation infrastructure. Considering that a controller needs to act on some components of the infrastructure that are owned by a third party, it is necessary that each party can trust each other in order to enable the negotiations.

Challenge In a federated authorisation infrastructure in which self-adaptation underpins the authorisation services, the challenge is the ability of establishing boundaries of awareness and influence and the ability of handling the dynamic nature of these boundaries.

Relevance Control boundaries are particularly relevant in federated authorisation infrastructures, where the controller does not have direct control over the third-party components. The controller may be able to request components of a federated authorisation framework to enact some changes, but these changes may only be accepted if they do not conflict with the goals of those components. In a federated authorisation infrastructure, boundaries can be related to what can be monitored and control, and to levels of trust, for example.

3.4.4 Execute

The Execute stage is responsible for executing the adaptation plan generated during the Plan stage. However, the execution of the plan may not always be straightforward since it involves several distinct needs, including the following ones:

- meet the objectives of the adaptation plan (including the synthesis of effectors, deployment of probes/gauges) and provide assurances that effectors have indeed carried out their actions (i.e. feedback of success);
- effectors must trust the controller of the self-adaptive authorisation infrastructure, in terms of authentication, authorisation, and non-repudiation;

- Execute stage is able to coordinate the effectors in issues, like concurrency, roll-backs and commits, recovery from failed plans, and heterogeneity of effectors;
- adaptation plans incorporate redundancies for making its execution more resilient and for supporting the provision of trust that resilience can be achieved;
- execution of the adaptation plan is secure in order to avoid exploitation of vulnerabilities;
- adaptation plans incorporate abstract commands since it should be down to the effector to decide how to implement a given action of the plan;
- synthesise and/or deploy probes, gauges, and effectors, or even update its own adaptation strategy in order to respond to new threats being detected;
- ability to reloading adapted authorisation policies or restarting authorisation services or other related services;
- ability to communicating with third-party identity providers, amongst other services, for example.

Underpinning all these needs, there is the fundamental need to provide assurances that the execution of the plan is according to its specifications. The execution of a plan may involve checking post-conditions, and it should provide feedback of its progress. In the following, some of the above needs will be detailed in terms of challenges.

3.4.4.1 Run-Time Synthesis of Effectors

It is not possible for the developers, at development time, to foresee all the possible threats that the system could face. Too many of those depend on the environment in which the system operates, and this environment can change at any time. The synthesis of new effectors at run-time allows the system to react to changes in the system or its environment that would otherwise affect the resilience of the self-adaptive authorisation infrastructure.

Challenge The challenge of a self-adaptive authorisation infrastructure to react to changes that are not foreseen at development time is to synthesise effectors at run-time.

Relevance Responding to threats that had not been foreseen during development time is essential for self-adaptive authorisation infrastructures. Occasionally, handling some of these threats may require effectors that are not yet available. In particular, the self-adaptive authorisation infrastructure may require new effectors for modifying the authorisation infrastructure. The ability to synthesise effectors during run-time will broaden the range of unforeseen threats that the system can protect itself against.

3.4.4.2 Deployment and Withdrawal of Probes, Gauges, and Effectors

Although the deployment and withdrawal of probes, gauges, and effectors might be outside the context of Execute stage of a self-adaptive authorisation infrastructure, these are an integral part of the adaptation plan and its execution. Probes, gauges, and effectors could either be taken from a pool, which is populated at development time, or synthesised at run-time for allowing the system to react to unforeseen changes. These dynamic deployment and withdrawal are different from the active monitoring challenge discussed in Sect. 3.4.1.1. In active monitoring, deployed probes and gauges have their own self-adaptive mechanism. In contrast, we discuss in this section the deploying and withdrawing of probes, gauges, and effectors as the result of the evolution of self-adaptive authorisation infrastructures. As such, in the deployment of new probes, gauges and effectors instead of being under the direct responsibility of a self-adaptive authorisation infrastructure, we could have a higher-level entity responsible for controlling the evolution of the self-adaptive authorisation infrastructure.

Challenge One of the challenges in self-adaptive authorisation infrastructures is the ability to deploy and withdraw probes, gauges, and effectors because of the wide range, and volatile nature, of threats that the system has to protect itself against.

Relevance In a self-adaptive authorisation infrastructure, changes affecting the infrastructure or its environment should be handled by the controller, and the ensuing adaptations may have an impact on how the controller observes and effects the infrastructure and/or its environment. Consequently, this may affect the probes, gauges, and effectors that are deployed. The complexity of the deployment can range from entirely predefined probes, gauges, and effectors that simply need to be activated to the synthesis of new ones. An intermediate solution would be the ability to configure predefined probes, gauges, and effectors for a particular use.

3.4.4.3 Trust

Trust is necessary between the parties in a federated authorisation infrastructure. The controller should trust that the other parties will carry out the plan as expected, and the other parties must trust that the controller acts in their advantage.

Challenge A key challenge in a self-adaptive authorisation infrastructure is to maintain trust between the parties by ensuring all parties behave as agreed. This is not restricted to techniques that ensure trust is maintained but also associated with strategies that are used when reacting to a breach of trust.

Relevance If a malicious user has been detected and reported to the identity service, but the identity service fails to take action to suspend the malicious user, then the trust between the authorisation infrastructure and the identity service should be reevaluated. As authentication is a critical component in access control, it is crucial for the authorisation infrastructure to be able to react to such breaches of trust, as they may harm the protected system.

3.4.4.4 Update or Redeployment of Policies and Sessions

Self-adaptive authorisation infrastructures can adapt authorisation policies in various ways. The adaptation could either take the form of an update of the current policy, where the controller sends the modifications to the policy decision point (PDP). Alternatively, the controller may create a whole new policy and instruct the PDP to deploy it instead of the previous one.

When adapting authorisation policies, the infrastructure should also consider the sessions that are currently open by a particular user and which may carry permissions that the user should not be assigned anymore sessions. All sessions could be revoked every time adaptation occurs. Alternatively, sessions could be amended in order to reflect the changes made in the authorisation policy.

Challenge During the execution of the plan, the challenge is to reduce the system vulnerability while policies are updated or redeployed.

Relevance The adaptation of an authorisation policy is an important part of a self-adaptive authorisation infrastructure's reaction to an internal threat. It is important that the adaptation is completed in a timely manner, in order to minimise the amount of time during which an attacker can cause damage to the system. The choice between updating the existing policy or deploying a new one should take into account the amount of time required for the new policy to be effective. Updating an existing policy requires the controller to communicate to the PDP only the changes to be made to the current policy. It is then the PDP's responsibility to enact those changes. Deploying a new policy, however, requires the controller to prepare a complete, updated policy, and to communicate in to the PDP, which only needs to deploy it to replace the previous one.

Existing sessions should also be taken care of in order to adapt the permissions given to the users that are logged on to the system while the adaptation takes place. Terminating all sessions is a simple solution, but it will require each use to authenticate again. In some circumstances this may not be ideal, especially if adaptation occurs often. The alternative is to modify user sessions at run-time, which may be more difficult to implement.

A similar challenge could be associated with the deployment of new probes and effectors, in particular those that are third party. The heterogeneity and the inflexibility of such devices may introduce vulnerabilities during adaptation.

3.4.4.5 Redundancy

Introducing redundancy in the Execute stage is one way to increase the resilience of the system. In case an effector fails to properly execute a portion of the plan, the plan should incorporate redundancies, using other effectors, alternative solutions, or workarounds, that would allow to tolerate the failed execution.

While the incorporation of redundancies in case of failure may be, in part, the responsibility of Plan stage, it is the responsibility of the Execute stage to monitor the execution of that plan and to trigger the redundancy measures when necessary.

Challenge The challenge is to include sufficient redundancies in the execution of the plan in order to make adaptation more resilient against potential threats, being these either internal attacks or faults.

Relevance The consequences of a failure during the execution of a plan while adapting an authorisation policy can be disastrous. At best, the old policy will still be in place, and the system will not be protected against the newly identified threat. At worst, the authorisation infrastructure could fail, either by locking all users out or by letting everyone access everything. Implementing redundant mechanisms to update policies will increase the service's resilience.

3.4.5 Models

This section on models refers to the Knowledge stage of the MAPE-K loop since models are the key source of knowledge in the self-adaptive authorisation infrastructure.

There are several types of models relevant to authorisation infrastructures, as well as many ways of using them for self-adaptation. In this section, we first categorise models for authorisation infrastructures into four types: authorisation polices, access control, threat, and adaptation. We then focus on the challenges that stem from the use of these models, which include portability, facilitating negotiation, history of models, uncertainty and conflicts in models, and model drift.

3.4.5.1 Modelling Authorisation Policies

Authorisation policies are a central component of authorisation infrastructures, where rules or assignments are defined and whose evaluation determines whether requests for access to protected assets are granted or denied. Policies can be very large, and they can be distributed across several documents, using various technologies, or, in the case of federated authorisation infrastructures, under the control of different entities.

Challenge Models of authorisation policies should be understandable yet precise for facilitating their manipulation by both the controller and system administrators.

Relevance For supporting the automated manipulation of models, these need to be precise. Moreover, these models need to be accessible to users in order to allow their validation against their respective policies. The reason is that, since authorisation policies determine the rendering of access control decisions, the authorisation policies models should be consistent with the actual authorisation

policies in order to avoid discrepancies between the authorisation infrastructure and its controller. So whatever changes are made on the authorisation policies, either by users or automated tools, these need to be accurately reflected on their corresponding models.

3.4.5.2 Modelling Access Control

Authorisation infrastructures often involve several key components that are connected for rendering access control decisions. Architectural models at the controller should be able to capture the components of an authorisation infrastructure and how these components are connected.

Challenge Architectural models should be dynamic because the infrastructure is expected to change, and these should be captured by the models. Such dynamic architectural models should be able to capture issues, such as the unavailability of components. In the context of federated authorisation infrastructures, architectural models should also be able to support access control and provide assurances of this ability.

Relevance In self-adaptive authorisation infrastructures, dynamic architectural models are essential for enabling the controller to handle changes affecting the infrastructure. Architectural models enable to analyse the consequence of threats to the infrastructure and investigate potential architectural solutions for mitigating those threats. For example, if an identity service fails to revoke credentials from subjects that are perceived as persistent attackers, the self-adaptive authorisation infrastructure may choose to disconnect that identity service from its infrastructure. However, before implementing that solution, the authorisation infrastructure may evaluate the impact of such a measure towards its users.

3.4.5.3 Modelling Threats

The purpose of self-adaptive authorisation infrastructures is to defend the system against threats. Since threats can change throughout the lifetime of an infrastructure, threat models should be dynamic, i.e. models that are able to change according to the threats to the infrastructure.

Challenge For reasoning about threats in self-adaptive infrastructures, a modelling language for expressing dynamic threat models is needed. In addition to representing threats, threat models should capture the likelihood of their occurrence, the potential harm they can cause to the infrastructure's assets, and which countermeasures can be taken to address them. If the self-adaptive infrastructure is capable of discovering previously unknown threats, then the threat model should be adaptable at run-time.

Relevance Threats, whether internal or external, are what self-adaptive authorisation infrastructures try to defend the system against. This can only be done if these infrastructures have a suitable model of threats and if these models cannot be adapted according to ever-changing threats a vulnerability will ensue.

3.4.5.4 Modelling Adaptation

The various models supporting self-adaptive authorisation infrastructures should take into account the run-time adaptation capabilities of the infrastructure. In particular, adaption models should represent what parts of the authorisation infrastructure can be adapted, how the adaptation can be executed and in which order, and when or under which conditions adaptation can happen.

Challenge There is the need to specify adaptation models that would be able to coordinate adaptations taking at different levels of an authorisation infrastructure and to identify what kind of assurances those models can provide.

Relevance Considering that both authorisation infrastructures and their environments are intrinsically dynamic, adaptation models should be related to the architectural models of the infrastructure, models of the authorisation policies, and threat models. Adaption models should also capture how the controller communicates with the authorisation infrastructure, which includes the monitoring and controlling of the infrastructure. If adaptation models do not relate to all models that enable the support of self-adaptation of authorisation infrastructures, then inconsistencies might arise regarding how self-adaptation is enacted.

3.4.5.5 Portability

Portability can take different forms in self-adaptive authorisation infrastructures. It could mean that the system can be deployed on various types of infrastructures. It could also mean that the system should be able to function in heterogeneous environments, where different subsystems can communicate with each other. The former requires some form of vertical transformation, where abstract elements can be concretised in various ways, depending on the underlying infrastructure. The latter requires some form of horizontal transformation, where models and data can be communicated between subsystems that use different representations.

Challenge The development of self-adaptive authorisation infrastructures can be made either vertically and horizontally portable. Vertically portable self-adaptive authorisation infrastructures can easily be redeployed on different environments or implementations of components. Horizontally portable self-adaptive authorisation infrastructures have components that may use different data formats and protocols but are still able to communicate with each other.

Relevance Federated authorisation infrastructures have components controlled by various entity systems they use. Components of these infrastructures may run different technologies or different versions of the same technology. They may also evolve and change at any time. Therefore, it is necessary for the self-adaptive authorisation infrastructure to be designed in a portable way.

3.4.5.6 Facilitating Negotiation

Federated authorisation infrastructures require several components working together. However, these components may be owned and controlled by different entities, who may have conflicting goals and interests. Therefore, it may be necessary for multiple components to negotiate a solution that satisfies all parties.

Challenge Models are required to facilitate negotiation between several components of a federated authorisation infrastructure. In self-adaptive authorisation architectures, these models should allow components to understand the consequences of the proposed changes on users' ability to use the system and should allow components to express agreement or disagreement with some of those changes.

Relevance Self-adaptive federated authorisation infrastructures must be able to handle negotiation between several of their components. They must be able to exchange proposed solutions to problems and indicate agreement or preferences. The solutions will contain models of the proposed changes, in order for the components to make informed decisions about the proposals they are presented with.

3.4.5.7 Capturing the History of Models

Capturing the history of models allows for the analysis of changes that happened in the past. The detection of long-running attacks, forensics analysis, and the detection of the entry point of an attacker all require access to historical data about the state of the system.

Challenge Self-adaptive authorisation infrastructures should be able to keep a history of the models that they maintain, in such a way that does not degrade performance yet allows for efficient analysis of past events. The ability to correlate changes to different models is especially important.

Relevance Attacks can be carried out over long periods of time. Hence, understanding them may require to analyse the past states of the self-adaptive authorisation infrastructure, both in order to detect and prevent attacks but also in order to identify patterns of suspicious behaviour over a long period of time.

3.4.5.8 Analysis Capabilities

The Analyse stage of a self-adaptive system depends on the available models in the knowledge base. If these models cannot completely reflect the reality they represent, the Analyse stage may not be able to always come to the correct conclusion or even come to a conclusion at all. Therefore, analysis on incomplete models may lead to uncertainty. If several analysis components are used, this may also lead to conflicting results.

Challenge Self-adaptive authorisation infrastructures should be able to deal with uncertainty and conflicts, and these may need to be encoded in the models.

Relevance Detection of insider threats is difficult because a smart malevolent insider will attempt to try and pass their usage as legitimate. Therefore, there is often no clear-cut distinction between legitimate and malicious users, making their detection difficult and ambiguous. Moreover, in a federated self-adaptive authorisation infrastructure, various Analyse stages may reach different conclusions, even when considering the same data.

3.4.5.9 Model Drift

Dynamic models are at risk of drift over time. Model drift is the progressive increase in the discrepancy between the model and what the model represents. In self-adaptive authorisation infrastructures, dynamic models are used to represent the target system, as well as its environment. If the models do not correctly reflect the reality, this may lead to suboptimal, or harmful, adaptation decisions.

Challenge In self-adaptive authorisation infrastructures, model drift should be avoided in order to reliably detect suspicious activity and identify malicious actors.

Relevance Attackers are likely to try and masquerade their actions as legitimate in order to escape security measures. Therefore, it is important for self-adaptive authorisation infrastructures to keep models that are very close to reality – even a small drift may be used by the attacker to cover their tracks.

3.5 Conclusions

The provision of self-adaptive authorisation infrastructures is a promising solution to protect systems against the dynamic nature of attacks and uncertainties associated with them, such as insider threats. In this chapter, we have presented how this could be achieved architecturally by separating the specification of policies (i.e. self-adaptive authorisation) from the enforcement of these policies (i.e. self-adaptive access control). We have also presented several technical challenges associated with the self-adaptation of authorisation infrastructures, which followed the stages of

the MAPE-K feedback control loop. Of course, the list of technical challenges should not be considered exhaustive since several of them were not included due to space constraints. Moreover, in addition to the identified technical challenges that are based on our experience in building and deploying self-adaptive of authorisation infrastructures [2, 4], one would expect new technical challenges to arise, depending on authorisation infrastructure and their deployment.

For presentation purposes, it was natural to follow the MAPE-K feedback control loop for identifying the technical challenges; however, questions may be asked about the appropriateness of MAPE-K loop when finding solutions to the wide range of identified challenges. Authorisation infrastructures are inherently complex since they can be geographically distributed, and this might require architectural solutions for the controller that might go beyond what the classical MAPE-K loop is able to offer [16, 56]. For example, if perpetual evaluations [55] are needed in order to obtain confidentiality, integrity, and availability assurances for computer-based resources, regarding the adaptations performed to the authorisation infrastructure, then new ways of enforcing separation of concerns are needed at the controller level. This of course will raise a new set of technical challenges that should be specific to the provision of assurances.

References

1. Axiomatics: Axiomatics policy server [Online], Available from: https://www.axiomatics.com/axiomatics-policy-server.html. Accessed 17 Jan 2014
2. Bailey, C.M.: Self-adaptive Authorisation Infrastructures. Ph.D. thesis, University of Kent (2015)
3. Bailey, C., Chadwick, D.W., de Lemos, R.: Self-adaptive authorization framework for policy based RBAC/ABAC models. In: Proceedings of the 2011 IEEE Ninth International Conference on Dependable, Autonomic and Secure Computing, DASC '11, pp. 37–44. IEEE Computer Society, Washington, DC (2011). https://doi.org/10.1109/DASC.2011.31
4. Bailey, C., Chadwick, D.W., de Lemos, R.: Self-adaptive federated authorization infrastructures. J. Comput. Syst. Sci. **80**(5), 935–952 (2014). http://www.sciencedirect.com/science/article/pii/S0022000014000154, Special Issue on Dependable and Secure Computing the 9th {IEEE} International Conference on Dependable, Autonomic and Secure Computing
5. Bailey, C., Montrieux, L., de Lemos, R., Yu, Y., Wermelinger, M.: Run-time generation, transformation, and verification of access control models for self-protection. In: Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2014, pp. 135–144. ACM, New York (2014). https://doi.org/10.1145/2593929.2593945
6. BBC: Credit card details on 20 million South Koreans stolen [Online] (Jan 2014), Available from: http://www.bbc.co.uk/news/technology-25808189. Accessed 5 Jan 2014
7. Benantar, M.: Access Control Systems: Security, Identity Management and Trust Models. Springer, New York (2005)
8. Bistarelli, S., Martinelli, F., Santini, F.: A formal framework for trust policy negotiation in autonomic systems: abduction with soft constraints. In: Proceedings of the 7th International Conference on Autonomic and Trusted Computing, ATC'10, vol. 6407, pp. 268–282. Springer, Berlin/Heidelberg (2010). http://dl.acm.org/citation.cfm?id=1927943.1927968

9. Booth, R., Brooke, H., Moriss, S.: WikiLeaks cables: Bradley Manning faces 52 years in jail [Online] (30 Nov 2010), Available from: http://www.theguardian.com/world/2010/nov/30/wikileaks-cables-bradley-manning. Accessed 5 Jan 2014
10. Brun, Y., Marzo Serugendo, G., Gacek, C., Giese, H., Kienle, H., Litoiu, M., Müller, H., Pezzè, M., Shaw, M.: Software engineering for self-adaptive systems. Engineering Self-Adaptive Systems Through Feedback Loops, pp. 48–70. Springer, Berlin/Heidelberg (2009). https://doi.org/10.1007/978-3-642-02161-9_3
11. Cappelli, D.M., Moore, A.P., Trzeciak, R.F.: The CERT Guide to Insider Threats: How to Prevent, Detect, and Respond to Information Technology Crimes, 1st edn. Addison-Wesley Professional, Upper Saddle River (2012)
12. Caputo, D., Maloof, M., Stephens, G.: Detecting insider theft of trade secrets. IEEE Secur. Priv. **7**(6), 14–21 (2009). https://doi.org/10.1109/MSP.2009.110
13. Chadwick, D.W., Otenko, A.: The PERMIS X.509 role based privilege management infrastructure. In: Proceedings of the Seventh ACM Symposium on Access Control Models and Technologies, SACMAT '02, pp. 135–140. ACM, New York (2002). https://doi.org/10.1145/507711.507732
14. Chadwick, D.W., Zhao, G., Otenko, S., Laborde, R., Su, L., Nguyen, T.A.: PERMIS: a modular authorization infrastructure. Concurr. Comput. Pract. Exp. **20**(11), 1341–1357 (2008). https://doi.org/10.1002/cpe.v20:11
15. Demchenko, Y., Gommans, L., Laat, C.: Extending role based access control model for distributed multidomain applications. In: Venter, H., Eloff, M., Labuschagne, L., Eloff, J., Solms, R. (eds.) New Approaches for Security, Privacy and Trust in Complex Environments, IFIP International Federation for Information Processing, vol. 232, pp. 301–312. Springer (2007). https://doi.org/10.1007/978-0-387-72367-9_26
16. de Lemos, R., Potena, P.: Chapter 14 – identifying and handling uncertainties in the feedback control loop. In: Mistrik, I., Ali, N., Kazman, R., Grundy, J., Schmerl, B. (eds.) Managing Trade-Offs in Adaptable Software Architectures. Morgan Kaufmann, pp. 353–367 (2017). ISBN 9780128028551, https://doi.org/10.1016/B978-0-12-802855-1.00014-9
17. de Lemos, R., Giese, H., Müller, H., Shaw, M., Andersson, J., Litoiu, M., Schmerl, B., Tamura, G., Villegas, N., Vogel, T., Weyns, D., Baresi, L., Becker, B., Bencomo, N., Brun, Y., Cukic, B., Desmarais, R., Dustdar, S., Engels, G., Geihs, K., Göschka, K., Gorla, A., Grassi, V., Inverardi, P., Karsai, G., Kramer, J., Lopes, A., Magee, J., Malek, S., Mankovskii, S., Mirandola, R., Mylopoulos, J., Nierstrasz, O., Pezzè, M., Prehofer, C., Schäfer, W., Schlichting, R., Smith, D., Sousa, J., Tahvildari, L., Wong, K., Wuttke, J.: Software engineering for self-adaptive systems: a second research roadmap. In: de Lemos, R., Giese, H., Müller, H., Shaw, M. (eds.) Software Engineering for Self-Adaptive Systems II. Lecture Notes in Computer Science, vol. 7475, pp. 1–32. Springer, Berlin/Heidelberg (2013). https://doi.org/10.1007/978-3-642-35813-5_1
18. Dobson, S., Denazis, S., Fernández, A., Gaïti, D., Gelenbe, E., Massacci, F., Nixon, P., Saffre, F., Schmidt, N., Zambonelli, F.: A survey of autonomic communications. ACM Trans. Auton. Adapt. Syst. **1**(2), 223–259 (2006). https://doi.org/10.1145/1186778.1186782
19. Garlan, D., Cheng, S.W., Huang, A.C., Schmerl, B., Steenkiste, P.: Rainbow: architecture-based self-adaptation with reusable infrastructure. Computer **37**(10), 46–54 (2004). https://doi.org/10.1109/MC.2004.175
20. Hellerstein, J.L., Diao, Y., Parekh, S., Tilbury, D.M.: Feedback Control of Computing Systems. Wiley, New York (2004)
21. Hu, V.C., Kuhn, D.R., Xie, T., Hwang, J.: Model checking for verification of mandatory access control models and properties. Int. J. Softw. Eng. Knowl. Eng. **21**(01), 103–127 (2011)
22. Hu, V.C., Schnitzer, A., Sandlin, K., Scarfone, K.: Guide to Attribute Based Access Control (ABAC) Definition and Considerations. NIST Special Publication (2013)
23. IBM: IBM Security Intelligence with Big Data [Online], Available from: http://www-03.ibm.com/security/solution/intelligence-big-data/. Accessed 20 July 2014
24. ITU-T Rec. X.509: The Directory: Authentication Framework. ISO/IEC 9594-8 (2000)

25. Janicke, H., Cau, A., Siewe, F., Zedan, H.: Dynamic access control policies. Comput. J. **56**(4), 440–463 (2013). https://doi.org/10.1093/comjnl/bxs102
26. Kalam, A.A.E., Benferhat, S., Miège, A., Baida, R.E., Cuppens, F., Saurel, C., Balbiani, P., Deswarte, Y., Trouessin, G.: Organization based access control. In: Proceedings of the 4th IEEE International Workshop on Policies for Distributed Systems and Networks, POLICY '03, pp. 120–131. IEEE Computer Society (2003). http://dl.acm.org/citation.cfm?id=826036.826869
27. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. Computer **36**(1), 41–50 (2003). https://doi.org/10.1109/MC.2003.1160055
28. Koutsonikola, V., Vakali, A.: LDAP: framework, practices, and trends. IEEE Internet Comput. **8**(5), 66–72 (2004). https://doi.org/10.1109/MIC.2004.44
29. Kramer, J., Magee, J.: Self-managed systems: an architectural challenge. In: 2007 Future of Software Engineering, FOSE '07, pp. 259–268. IEEE Computer Society, Washington, DC (2007). https://doi.org/10.1109/FOSE.2007.19
30. Lopez, J., Oppliger, R., Pernul, G.: Authentication and authorization infrastructures (AAIS): a comparative survey. Comput. Secur. **23**(7), 578–590 (2004). https://doi.org/10.1016/j.cose.2004.06.013
31. McGraw, R.: Risk-adaptable access control (RADac). Technical report, National Institute of Standards and Technology (NIST) (2009)
32. Moore, A.P., Hanley, M., Mundie, D.: A pattern for increased monitoring for intellectual property theft by departing insiders. Technical report, CMU/SEI-2012-TR-008, Software Engineering Institute, Carnegie Mellon University, Pittsburgh (2012)
33. Morgan, R.L., Cantor, S., Carmody, S., Hoehn, W., Klingenstein, K.: Federated security: the Shibboleth approach. EDUCAUSE Q. **27**(4), 12–17 (2004). http://www.eric.ed.gov/ERICWebPortal/detail?accno=EJ854029
34. Mu, C., Li, Y.: An intrusion response decision-making model based on hierarchical task network planning. Expert Syst. Appl. **37**(3), 2465–2472 (2010)
35. NIST: INCITS 359-2004 – Role Based Access Control (2004)
36. Nurse, J.R., Buckley, O., Legg, P.A., Goldsmith, M., Creese, S., Wright, G.R., Whitty, M.: Understanding insider threat: a framework for characterising attacks. In: Workshop on Research for Insider Threat (WRIT) Held as Part of the IEEE Computer Society Security and Privacy Workshops (SPW14), in conjunction with the IEEE Symposium on Security and Privacy (SP), pp. 214–228. IEEE (2014). http://www.sei.cmu.edu/community/writ2014/
37. OASIS: Security Assertion Markup Language (SAML) Version 2.0 (2005)
38. OASIS: eXtensible Access Control Markup Language (XACML) v3.0 (2013)
39. O'Conner, A.C., Loomis, R.J.: 2010 economic analysis of role-based access control. Technical report, RTI International, NIST (2010)
40. Oltsik, J.: The 2013 Vormetric insider threat report [Online] (2013), Available from: http://www.vormetric.com/sites/default/files/vormetric-insider-threat-report-oct-2013.pdf. Accessed 12 June 2014
41. Oreizy, P., Gorlick, M.M., Taylor, R.N., Heimbigner, D., Johnson, G., Medvidovic, N., Quilici, A., Rosenblum, D.S., Wolf, A.L.: An architecture-based approach to self-adaptive software. IEEE Intell. Syst. **14**(3), 54–62 (1999). https://doi.org/10.1109/5254.769885
42. Park, J., Sandhu, R.: The UCONABC usage control model. ACM Trans. Inf. Syst. Secur. **7**(1), 128–174 (2004). https://doi.org/10.1145/984334.984339
43. Pashalidis, A., Mitchell, C.J.: A taxonomy of single sign-on systems. In: Proceedings of the 8th Australasian Conference on Information Security and Privacy, ACISP'03, pp. 249–264. Springer, Berlin/Heidelberg (2003). http://dl.acm.org/citation.cfm?id=1760479.1760507
44. Pasquale, L., Menghi, C., Salehie, M., Cavallaro, L., Omoronyia, I., Nuseibeh, B.: Securitas: a tool for engineering adaptive security. In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, FSE '12, pp. 19:1–19:4. ACM, New York (2012). https://doi.org/10.1145/2393596.2393618
45. Pearlman, L., Welch, V., Foster, I., Kesselman, C., Tuecke, S.: A community authorization service for group collaboration. In: Proceedings of the 3rd International Workshop on Policies

for Distributed Systems and Networks (POLICY'02), pp. 50–59. IEEE Computer Society, Washington, DC (2002). http://dl.acm.org/citation.cfm?id=863632.883495
46. PERMIS Standalone Authorisation Server: [Online], Available from: http://sec.cs.kent.ac.uk/permis/. Accessed 5 Jan 2014
47. Ratha, N.K., Bolle, R.M., Pandit, V.D., Vaish, V.: Robust fingerprint authentication using local structural similarity. In: Fifth IEEE Workshop on Applications of Computer Vision, 2000, pp. 29–34. IEEE (2000). http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.19.8588&rep=rep1&type=pdf
48. Serrano, M., Meer, S., Strassner, J., Paoli, S., Kerr, A., Storni, C.: Trust and reputation policy-based mechanisms for self-protection in autonomic communications. In: Proceedings of the 6th International Conference on Autonomic and Trusted Computing, ATC '09, pp. 249–267. Springer, Berlin/Heidelberg (2009). https://doi.org/10.1007/978-3-642-02704-8_19
49. SimpleSAMLphp: [Online], Available from: http://simplesamlphp.org/. Accessed 5 Jan 2014
50. Spitzner, L.: Honeypots: catching the insider threat. In: Proceedings of the 19th Annual Computer Security Applications Conference, pp. 170–179. IEEE (2003)
51. Stakhanova, N., Basu, S., Wong, J.: A cost-sensitive model for preemptive intrusion response systems. In: AINA. vol. 7, pp. 428–435 (2007)
52. Strasburg, C., Stakhanova, N., Basu, S., Wong, J.S.: A framework for cost sensitive assessment of intrusion response selection. In: Proceedings of the 2009 33rd Annual IEEE International Computer Software and Applications Conference, COMPSAC '09, vol. 01, pp. 355–360. IEEE Computer Society, Washington, DC (2009). https://doi.org/10.1109/COMPSAC.2009.54
53. Thompson, M., Johnston, W., Mudumbai, S., Hoo, G., Jackson, K., Essiari, A.: Certificate-based access control for widely distributed resources. In: Proceedings of the 8th Conference on USENIX Security Symposium, SSYM'99, pp. 17–30. USENIX Association, Berkeley (1999). http://dl.acm.org/citation.cfm?id=1251421.1251438
54. Walsh, C.: New data theft scandal rocks subcontinent's call centres [Online] (3 Sept 2006), Available from: http://www.theguardian.com/money/2006/sep/03/business.india. Accessed 5 Jan 2014
55. Weyns, D.: Software engineering of self-adaptive systems: an organised tour and future challenges. In: Cha, S., Taylor, R.N., Kang, K.C. (eds.) Handbook of Software Engineering. Springer, Cham (2018)
56. Weyns, D., Malek, S., Andersson, J.: Forms: unifying reference model for formal specification of distributed self-adaptive systems. ACM Trans. Auton. Adapt. Syst. **7**(1), 8:1–8:61 (2012). https://doi.org/10.1145/2168260.2168268
57. Yuan, E., Tong, J.: Attributed based access control (ABAC) for web services. In: Proceedings of the IEEE International Conference on Web Services, ICWS '05, pp. 561–569. IEEE Computer Society, Washington, DC (2005). https://doi.org/10.1109/ICWS.2005.25
58. Yuan, E., Malek, S., Schmerl, B., Garlan, D., Gennari, J.: Architecture-based self-protecting software systems. In: Proceedings of the 9th International ACM Sigsoft Conference on Quality of Software Architectures, pp. 33–42. ACM (2013)
59. Yuan, E., Esfahani, N., Malek, S.: A systematic survey of self-protecting software systems. ACM Trans. Auton. Adapt. Syst. **8**(4), 17:1–17:41 (2014). https://doi.org/10.1145/2555611

Chapter 4
Bidirectional Transformations for Self-Adaptive Systems

Lionel Montrieux, Naoyasu Ubayashi, Tianqi Zhao, Zhi Jin, and Zhenjiang Hu

Abstract Bidirectional transformations are a synchronisation mechanism between documents, a source, and a view. A bidirectional transformation is a pair of functions, one that extracts a view from a source and the other that updates a source according to changes made to the view. Bidirectional programming is a recent technique that helps developers to easily write bidirectional transformations and ensure that they satisfy properties of interest. In this chapter, we argue that bidirectional transformations and bidirectional programming are useful techniques in the context of self-adaptive systems. We present four applications of bidirectional transformation for construction of adaptive systems: abstraction, separation of concerns, rule-based adaptation, and uncertainty-aware programming.

4.1 Introduction

Bidirectional transformations [6, 11, 18] have been the focus of a lot of attention lately, both in the programming language community [2, 10, 12, 16, 26] and in the software engineering community [28, 29]. They are a recent way of solving the old view-update problem, defined decades ago in the database community. As bidirectional programming languages are growing more mature, they are getting easier to use for software engineers and more efficient and more reliable. Perhaps the strongest argument in favour of bidirectional programming is its ability to provide a synchronisation mechanism between a source and a view that is guaranteed to be correct *by construction*.

L. Montrieux (✉) · Z. Hu
National Institute of Informatics, Tokyo, Japan
e-mail: lionel.montrieux@zalando.de

N. Ubayashi
Kyushu University, Fukuoka, Japan

T. Zhao · Z. Jin
Peking University, Beijing, China

Y. Yu et al. (eds.), *Engineering Adaptive Software Systems*,
https://doi.org/10.1007/978-981-13-2185-6_4

In this chapter, we present four different ways in which bidirectional programming and bidirectional transformations can improve the state of the art in engineering self-adaptive systems. In particular, we focus on self-adaptive systems developed around the MAPE-K feedback loop model [19].

First, bidirectional programming can be used to synchronise concrete and abstract models in the knowledge base, allowing developers to write their adaptation layer independently of the implementation of the target system. This facilitates the reuse of MAPE-K components and allows developers to easily swap the implementation of the target system, without having to rewrite the adaptation layer.

Second, bidirectional programming is useful to achieve separation of concerns, and hence reuse of components, in the adaptation layer. By extracting from the knowledge base small models that are tailored to a particular aspect being analysed, bidirectional transformations simplify the development of small, focused analysis and planning components and may even improve performance, due to the reduced size of the models to consider.

Third, bidirectional programming is used in the context of view-based adaptation rules for the analysis of the target system and its environment. Bidirectional programming comes as a natural approach to implement the νRule approach, where adaptation rules are applied depending on the state of the environment, captured in views.

Fourth, bidirectional programming is applied in the field of uncertainty-aware software development, a software development approach that makes uncertainty a first-class citizen. Bidirectional transformations can extract partial models from code and uncertainty-aware artefacts and reflect changes made to the partial models, back to the code.

The rest of this chapter is organised as follows: In Sect. 4.2, we introduce bidirectional transformations and bidirectional programming. In Sect. 4.3, we discuss how bidirectional programming can synchronise concrete and abstract models, and in Sect. 4.4, we focus on separation of concerns. Section 4.5 discusses the use of bidirectional programming with νRule -based adaptation. Then, Sect. 4.6 considers the role of bidirectional programming in the context of uncertainty-aware development. Section 4.7 concludes this chapter.

4.2 Bidirectional Programming

A bidirectional transformation is a pair of functions, a forward transformation `get` and a backward transformation `put`, used to synchronise two documents [11]. The forward transformation takes a source as input and produces a view; the backward transformation takes a source and a view as inputs and uses the view to update the source, producing an updated source.

In this paper, we will use Haskell, a functional language, to specify bidirectional transformation. One big reason for us to choose Haskell is that a set of bidirectional languages (libraries) have been developed in Haskell.

In Haskell, the types for `get` and `put` are the following for a source of type `Source` and a view of type `View`:

```
get :: Source -> View
put :: Source -> View -> Source
```

For example, the following code defines a bidirectional transformation between a list of integers (the source) and a single element (the view).

```
1 get :: [Int] -> Int
2 get (x:xs) = x
3
4 put :: [Int] -> Int -> [Int]
5 put (x:xs) y = y:xs
```

The `get` function extracts the head of the list, while the `put` function updates the head of the source list with the value in the view, as illustrated by the following example:

```
> get [1,2,3]
1

> put [1,2,3] 9
[9,2,3]
```

A particularly interesting class of bidirectional transformations are *well-behaved* bidirectional transformations [11, 12]. Intuitively, a well-behaved bidirectional transformation provides a "correct" synchronisation between source and view. More formally, a bidirectional transformation is well behaved if it satisfies two properties: `GetPut` and `PutGet`.

`GetPut` is the identity law. If a view is extracted from a source and used and unchanged to update the source, the source should not change:

```
put s (get s) = s
```

`PutGet` is the change conservation law. If a view has been updated, then using it to update the source and then extracting a view from that updated source should produce the same updated view:

```
get (put s v) = v
```

The example we used above is a well-behaved bidirectional transformation. Using the same source as above, both laws of well-behaved bidirectional transformations are satisfied:

```
> put [1,2,3] (get [1,2,3]) = put [1,2,3] 1 = [1,2,3]

> get (put [1,2,3] 99) = get [99,2,3] = 99
```

Our example is trivial, but it can be very difficult to write a complex bidirectional transformation, let alone proving it well behaved. Still, bidirectional transformations

have been used in a variety of applications [6], including spreadsheets [5], graph transformations [16], and many more.

Bidirectional programming languages are domain-specific languages that aim to simplify the development of bidirectional transformations [11]. Developers write one direction of the transformation, and the bidirectional programming language's compiler automatically derives the other direction, to form a well-behaved bidirectional transformation, if it exists.

We can classify these bidirectional programming languages in two categories: *get-based* languages and *put-based* (also called *putback-based*) languages.

Get-based languages let developers write a `get` function and automatically derive a `put` function, forming a well-behaved bidirectional transformation. GroundTRam [17] is one of these languages. The advantage of get-based languages is that the `get` function is relatively easy to write. The inconvenience is that for a given `get` function, there may be many `put` functions that form a well-behaved bidirectional transformation. For example, the following code snippet shows three different `put` functions for a single get. They all form well-behaved bidirectional transformations:

```
1  get :: [Int] -> Int
2  get (x:xs) = x
3
4  put1 :: [Int] -> Int -> [Int]
5  put1 (x:xs) y = y:xs
6
7  put2 :: [Int] -> Int -> [Int]
8  put2 (x:xs) y = if x==y then y:xs else y:[]
9
10 put3 :: [Int] -> Int -> [Int]
11 put3 x y = if x>=y then y:xs else y:[]
```

Put-based languages, on the other hand, let developers write a `put` function and automatically derive a `get` function to form a well-behaved bidirectional transformation. While the `put` function is often more difficult to write than the `get` function, we know that for a given `put`, there exists *at most one* `get` that forms a well-behaved bidirectional transformation [9]. Hence, put-based languages give developers more control over their bidirectional transformations. Examples of put-based languages include BiFluX [25], BiGUL [21], or Brul [30].

Bidirectional programming, like bidirectional transformations, has been applied to a variety of areas in software engineering, such as access control [13, 24], model-code synchronisation [29], or self-adaptive systems [4].

4.3 Bidirectional Programming and Abstraction

In self-adaptive systems built around the MAPE-K loop [19], the adaptation layer monitors the target system and its environment, analyses data, plans changes, and executes changes on the target system. Often, the self-adaptive layer is decoupled from the target system. The monitoring of the system is done through probes and gauges and the execution through effectors. This decoupling makes it easy to reuse the same self-adaptive layer for multiple target systems. In this section, we show how bidirectional programming can facilitate the reuse of a self-adaptive layer for multiple target systems *independently* of the implementation of the target system [4]. In particular, we focus on the adaptation of configuration files, which is a common way of controlling the behaviour of the target system.

Configuration files are trees. They contain key-value pairs, each used to configure a particular aspect of the software. In some configuration files, these pairs can be placed inside blocks, and blocks can contain other blocks. Depending on the configuration file, the order of pairs and/or blocks may matter. Keys that do not appear in the configuration file are given a default value when parsed by the software. The use of blocks also allows for context overriding, where the value of a key in a block takes, for that block, precedence over other values of the same key defined in ancestor blocks or over the default value.

An example of software that uses configuration files is a web server. There are many implementations of web servers, such as Apache, Nginx, or Microsoft IIS. Each implementation defines its own configuration format, syntax, and semantics. Yet there are a lot of similarities between each implementation's configuration files. After all, they all describe the behaviour of web servers.

Using bidirectional programming, it is possible to synchronise a concrete configuration file with an abstract web server configuration that is independent of the implementation used. For each implementation, a bidirectional transformation synchronises the concrete configuration file (the source of the transformation) with the abstract configuration (the view), as depicted on Fig. 4.1. Since the transformations

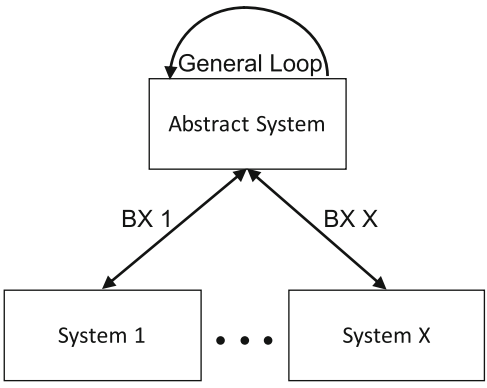

Fig. 4.1 Abstraction with bidirectional programs

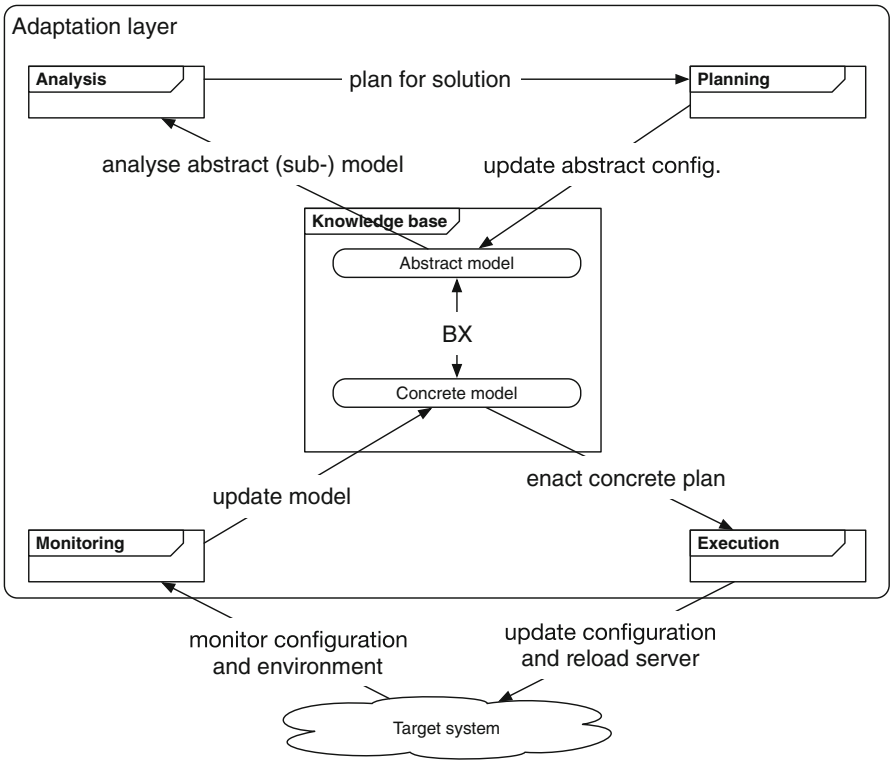

Fig. 4.2 MAPE-K loop over abstract configuration

for all implementations use the same type for the view, it is then possible to reuse the abstract implementation when implementing the MAPE-K loop, as illustrated on Fig. 4.2. The monitoring stage keeps track of the environment and of changes in the software's concrete configuration. Changes in the configuration trigger a new `get` transformation that updates the abstract configuration. Both the analysis and the planning stages use the abstract configuration and are therefore reusable across any implementation. The planning stage can directly modify the abstract configuration to enact adaptation. In the execution phase, a `put` transformation synchronises the abstract configuration with the concrete configuration, before the configuration file is transferred to the target system, and the target system restarted or reloaded, if necessary.

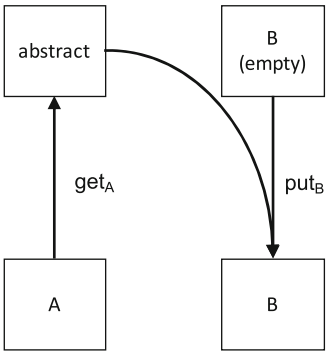

Fig. 4.3 Migration with bidirectional programs

4.3.1 Migration

An additional benefit of using bidirectional transformations to synchronise concrete and abstract configurations is the possibility to migrate the target system from one implementation to another while conserving the same configuration. Figure 4.3 illustrates such a scenario. Let A and B be two implementations of the target system, each with its own configuration format. Two bidirectional programs are written to synchronise the concrete configurations with a common abstract configuration. To migrate the target system from implementation A to implementation B, an abstract configuration is first extracted from the concrete configuration of A, using the `get` transformation provided by the bidirectional program for A. Then, the abstract configuration is used, together with an empty template configuration for B, to produce, through the `put` transformation provided by the bidirectional program for B, a concrete configuration for B that captures the same behaviour as the concrete configuration for A.

4.4 Bidirectional Programming and Separation of Concerns

In the previous section, we discussed how to apply bidirectional programming to easily develop synchronisation between abstract and concrete models in the knowledge base. This synchronisation, correct by construction, allows for the reused of parts of the adaptation layer, regardless of the implementation of the target system. This is not the only way in which bidirectional programming can be helpful in the knowledge base. It can be also used to facilitate separation of concerns in adaptation layers that adapt a target system according to multiple concerns.

For example, the adaptation of a web application deployed on an IaaS cloud service could take into account the usage of the cloud instances, security concerns, and service availability requirements. While it is possible to consider these concerns together as a multi-objective optimisation problem, we argue that considering them separately has benefits and that bidirectional programming helps to develop such systems.

4.4.1 Extracting Submodels

In self-adaptive systems, the analysis of a particular concern may only require a subset of the entire model(s)[1] in the knowledge base. Using a bidirectional program, it is possible to extract the exact model subset necessary to perform adaptation according to a given concern and to keep it synchronised with the whole model. In this situation, the complete model is the source, while a submodel for a given concern is a view. The submodel can be analysed and then passed on to the planning phase, which can directly modify the submodel. A `put` transformation will ensure that the source model is updated accordingly. Figure 4.4 illustrates this process.

There are several advantages in extracting a submodel for each concern:

- The phases using the submodel do not need to change if unrelated changes happen in the complete model. It is therefore easier to implement reusable analysis and planning phases;
- If the analysis and/or planning phases use techniques whose performance degrades with large inputs, such as model checking, a comparatively small submodel can speed up the adaptation;
- Once views are extracted, it is easy to parallelise the execution of the analysis and planning phases of each concern.

One inconvenient of this approach is that two separate concerns may cause conflicting modifications of the source model. When evaluating the entire source at once, mitigation strategies could easily be employed. Our approach would make this more complicated. Ordering can be used to favour the more important concerns

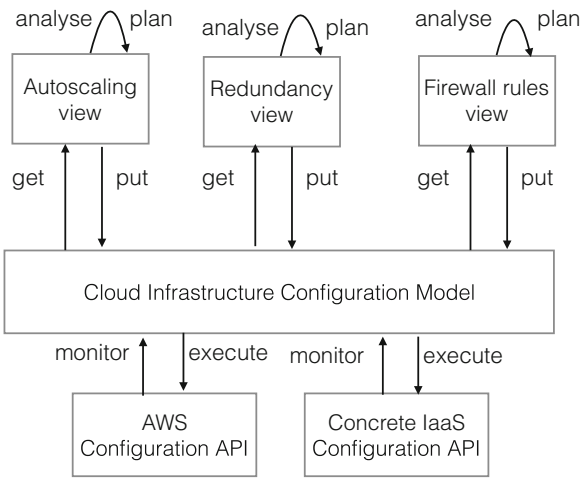

Fig. 4.4 Extracting a submodel

[1]Since multiple models can easily be merged into a single one using a virtual root element, we assume from now on that the knowledge base contains only one model.

over the less important ones. In the case of several, equally important and potentially conflicting concerns, separation may cause difficulties. However, it may still be possible to group these concerns into a single view and perform the analysis for these concerns simultaneously on a model that is still smaller than the source.

Using bidirectional programs provides guarantees, by construction, of the correctness of the synchronisation mechanism between the source and each of the views. In our approach, only the analysis and planning phases use a view. The monitoring phase directly updates the source, and the execution phase uses data from the source, after it has been modified by all the planning phases. In the spirit of Weyns et al.'s MAPE-K patterns for distributed systems [27], our solution follows a M(AP) + E pattern.

4.4.2 Current vs. Desired State of the Model

In addition to using views to achieve separation of concerns, bidirectional programming offers a solution to the issue of the model's state. The solution is to use two views, one for the current state of the model and one for the desired state of the model. Both views are derived from the same source. Components can use the most appropriate view.

In the source, a new status field is added to each element in the model. The status indicates whether the element should not change (`0`), be created (`1`), or be deleted (`2`) from the current model. Modifications to an element are treated as both a deletion and a creation. The `put` transformation for the current model always sets that value to `0` in the source. However, the `put` transformation for the desired model sets the status to `1` for all elements created in the source and sets it to `2` for all deletions, without actually deleting the element from the source. The `get` transformations derived are simple: for the current model, it selects all elements whose status is `0` or `2`; for the desired model, it selects all elements whose status is `0` or `1`.

4.4.3 Evaluation Order and Concurrent Evaluation

The concurrent evaluation of multiple views can cause consistency issues. Let two views, V_0^1 and V_0^2, extracted from the same version of the source S_0 (i.e. there has been no `put` transformation run between the extraction of the two sources), using `get` transformations BX^1 and BX^2, respectively. The respective analysis and planning phases for V_0^1 and V_0^2 could make any changes to the views, resulting in V_1^1 and V_1^2, respectively. Propagating these changes to the source must be done sequentially, as languages like BiGUL do not support the simultaneous update of a source using multiple views. We assume that the transformation using V_1^1 is performed first, but the argument is valid the other way around. Using BX^1, a `put` transformation updates S_0, which becomes S_1. If a `get` transformation with BX^2

still produces V_0^2, then the other view has not been affected, and the second `put` transformation can be performed. Otherwise, the second view is based on outdated data, and it is possible that, should the analysis and planning be run again, another result would be reached.

There are several ways to solve this issue. Perhaps the simplest approach is to define a partial order between the different concerns and deal with them sequentially according to the ordering (i.e. for each concern, start with `get` and then analyse and plan and finish with `put`). Another strategy is, at design time, to manually inspect the views produced and run those views that are produced from entirely distinct subsets of the source in parallel, as they are completely separate. Finally, a more sophisticated approach would be to deal with several concerns in parallel, keeping track of the view produced by each `get` transformation (i.e. before it is modified). After every `put`, all `get` transformations would be run again. If they produce the same view as the previous `get` transformation, then the view has not been affected by the changes introduced by `put`, and the updated view is still valid. Otherwise, the analysis and planning must be run again. In the worst case, this will be equivalent to the sequential scenario, with the addition of the comparison between different versions of the views.

4.5 Declarative Description of Adaptation Logic

Adaption logic plays an important role in self-adaptive systems, specifying when and how to update the behaviour and/or structure of the system in response to changes of the environment and the system itself. So far, it has been implemented mostly by hardwired code for analysing and planning in the MAPE-K loop. In this section, we show that adaptation logic can be declaratively specified in putback-based bidirectional languages such as BiFluX and BiGUL, which would allow developers to utilise bidirectional programming to systematically construct robust self-adaptive systems.

4.5.1 Adding Views to Adaptation Rules

Rule-based adaptation [1, 7, 22] provides a powerful mechanism to develop self-adaptive systems, enabling systems to modify their behaviour, reconfigure their structure, and evolve over time, reacting to changes in the operating context [3]. In a rule-based adaptation system, a set of adaptation rules are used to specify adaptation logic of what particular action should be performed in reaction to monitored events.

Typically, an adaptation rule takes the form of *condition* $\Rightarrow$ *action* where *condition* specifies the trigger of the rule, which is often fired as a result of a set of monitoring operations, and *action* specifies an operation sequence to perform

in response to the trigger. For instance, in a smart room system, we may have the following rule:

$$Light.Power = off \ \wedge \ Time = daytime \\ \Rightarrow Blind.State := open; \ Window.State := open$$

which declares that if the light is powered off in daytime, then the blind and the window must be opened. Rule-based adaptation has advantages of readability and elegance of each individual rule, the efficiency of plan process and the ease of rule modification.

In spite of these advantages, adaptation rules pay attention only to local transformations. However, such local transformations can be structured to ease the satisfaction of the system's global goals. In many cases, some environment features may hint at different system goals, and different system goals imply different adaptation policies.

We introduce another concern, *situation*, for capturing such a hint [20]. With this concern, the adaptation logics are of two layers. The first layer intends to capture the requirement changes when the situation changes. The second layer intends to capture, for a certain system goal, the situation changes or changes in some of the entities in the environment that require actions to continue satisfying the system goal.

Then, to enable the dynamic decision on the system adaptation, three types of νRules can be identified:

- situation $\rightarrow$ goal setting: It captures the phenomena that the user may have different desires in different situations.
- goal setting: situation $\rightarrow$ behaviour pattern: This means that, given a goal setting, the system should behave according to different behaviour patterns when situated in different situations. A behaviour pattern consists of a set of environment features.
- goal setting: environment features $\rightarrow$ system features: This means that, given a goal setting, some of the system features need to be enabled by current emerged environment features to better satisfice the goal setting.

The first type of rules is meaningful in decision-making about adaptation. For example, *everybody is sleeping* is a situation that represents "everybody has been in bed". When in this situation, the system normally switches to the goal setting of the "sleeping mode" without taking into account other environment features.

When a goal setting needs to be continuously satisfied, different situations may also indicate different system behaviour modes. This is represented by the second type of rules. In fact, situation is a concept that has received much attention from philosophers and logicians. The earlier formal notion of situation was introduced as a means to give a more realistic formal semantics for speech acts than what was then available. In contrast with a *world* which determines the value of every proposition, a situation corresponds to the limited parts of reality we perceive, reason about, and live in. With limited information, a situation can provide answers to some, but

not all, questions about the world. The advantage of including situation is then to decrease the sensing cost as, normally, only parts of the environmental setting need to be detected when making a decision in a particular situation. This is important when the number of environment entities is large. The other advantage could be fitting to the human recognition. In many cases, only a few features are required to identify a situation, while others are less important.

4.5.2 *vRule: View-Based Adaptation Rule*

We assume that the environment states, the goal settings and the system behaviours are represented by feature bindings and propose to structure adaptation rules into vRules. There are two types of rules. The first type of vRule is the behaviour rule. It is made of three parts: an observable view (v) that could be a goal configuration, a conjunction of conditions (C) that could be a situation of the environment (a group of significant environment features that indicates the situation) or a set of environment features (that does not indicate a situation but captures the environment states) and a sequence of actions (A). The second type of view-deciding rule is made of two parts: an observable situation of the environment (C) and a system goal configuration (A). For unification, we assume the view part of the second type of rule is $true$.

The concrete syntax of vRule is shown in Fig. 4.5.

A vRule

$$v \vdash C \Rightarrow A$$

can be read as "if v holds, action A should be taken under condition C and preserve state v". The view v and the condition C are defined over feature bindings, where a feature is a goal setting or an environment attributes or a system component. A feature binding fb has two alternatives: *feature* can be either bound with a *value* or a *value interval*. For example, $Light.Intensity = 2$' means the intensity of the light is 2, $Light.Intensity = (1, 3]$ equals to $1 < Light.Intensity \leq 3$.

A is a set of asked system component settings a, and each setting a binds a constant *value* to a *feature*. For example, in the following vRule

Fig. 4.5 Syntax of vRule

view-based rule	$\mathsf{v}Rule$	$::= v \vdash C \Rightarrow A$
view	v	$::= fb$
feature binding	fb	$::=$ *feature* = *value*
		$\mid$ *feature* = *value interval*
conditions	C	$::= c_1 \wedge c_2 \wedge \ldots \wedge c_m$
condition	c	$::= fb$
action sequence	A	$::= a_1; a_2; \ldots; a_n$
action	a	$::=$ *feature* := *value*

$$(r_1)\ \mathit{Light.Power} = \mathit{off}$$
$$\vdash \mathit{Time} = \mathit{daytime}\ \wedge \mathit{Blind.State} = \mathit{close}$$
$$\Rightarrow \mathit{Blind.State} := \mathit{open};\ \mathit{Window.State} := \mathit{open}$$

r_1 declares that when the current goal is to keep the light powered off, if it is in daytime and the blind is closed, then the system components, the blind, and the window will be opened.

Generally, the νRule implies that the *view* needs to be kept after adaptation. That is the reason of calling the rule the *view-based* adaptation rule. *The key is the use of the idea of "view" in the rule specification. Rather than showing how to propagate changes (out), the* view-based *rules specify how a view can be kept through changes of necessary system components for responding the environment changes.*

In the condition of a νRule, we do not support the *or* operation. This does not weaken the expressiveness of νRules, as a νRule with a c_1 *or* c_2 condition is equivalent to two νRules, with conditions c_1 and c_2, respectively.

4.5.3 Implementation of vRule in BiGUL

The proposed νRule s can be implemented in BiGUL. The implementation of νRule by BiGUL includes two parts: (1) representation of the view and the source models and (2) translation of νRule s as BiGUL updates.

4.5.3.1 Representation

For a specific νRule, the view model only specifies the binding state of one feature, and the source model includes the binding states of all features monitored from the environment and system. Therefore, we represent the view as a feature binding, which is a tuple consisting of the feature name and its value, and represent the source as a set of feature bindings. Figure 4.6 gives an example of the source and the view for νRule $r1$, in the sense that the view is a projection of the source, where only the value of feature "Light.power" is considered.

4.5.3.2 Translation

With the source and the view represented, a νRule can be translated into updates in BiGUL. We describe the translation from νRule to BiGUL by using the νRule $r1$ as an example. The translated BiGUL program is as follows (Fig. 4.7):

This program means updating the source using the view: if the view feature "Light.power" takes the value "off", then the source feature model will be updated with the value of feature "Blind.state" set to "on", and the value of feature "Window.state" set to "open".

```
s :: [(String, String)]
s = [
     ("aircon_power","off"), ("light_power","off"), ("light_intensity","off"),
     ("curtain","open"), ("window_state","open"), ("setpoint","no"),
     ("somebody_home","true"), ("family_sleep","true"), ("weather","sunny"),
     ("time","daytime"), ("brightness","10"), ("temperatue","10"),
     ("room_temperatue","10"), ("room_brightness","10"), ("blind_state","open"),
     ("cdplayer_power","on"), ("light_style","gentle")

   ]

v :: [(String, String)]
v = [
     ("aircon_power","on"), ("light_power","on")
   ]
```

Fig. 4.6 An example of the source and the view

```
rule1 = Case [
          $(adaptive [| \s v -> v == ("Light_power","off") &&
                              s !! ("Time","daytime") &&
                              s !! ("Blind_state", "close") &&
                        \s v -> set' s [("Blind_state","on"), ("Window_state", "open")]],
          $(normal [| \s v -> True |]) $
                rep_i (list_id "Blind_state")
           ]
```

Fig. 4.7 The BiGUL program of νRule r1

Fig. 4.8 Consistency check

```
checkEqual r1 r2 s v =
  case (put2rules [r1, r2] s v, put2rules [r2, r1] s v)
    (Right leftS, Right rightS) ->
      if leftS == rightS then True
      else False
```

A νRule is regarded as valid when it satisfies the view preservation property: if the *view* holds, it should still hold after the execution of the action. Implementing a νRule as a BiGUL program can facilitate the check of its validity. A BiGUL program is guaranteed to produce a well-behaved bidirectional transformation if one exists, i.e. a BiGUL program satisfies the `GetPut` and `PutGet` laws (see Sect. 4.2). Therefore, a successfully compiled BiGUL program is guaranteed to be view preserved, and in this case, the corresponding νRule is guaranteed to be a view-preserved rule.

While a νRule is valid when it is view preserved, a νRule set is regarded as well behaved only if (1) each νRule in this set is valid and (2) every two rules in this set are order independent. While the validity of a single νRule can be automatically checked, the order independence between two rules can be checked through the "`checkEqual`" function in Fig. 4.8. For two νRule s $r1$ and $r2$, if executing $r2$ after $r1$ and executing $r1$ after $r2$ lead to the exactly same adaptation results, $r1$ and $r2$ will be considered as order independent.

4.6 Bidirectional Transformations for Uncertainty-Aware Software Development

4.6.1 Uncertainty in Software Development

Recently, uncertainty has attracted a growing interest among researchers. Research themes spread over uncertainty of goal modelling, UML modelling, model transformations, and testing. Garlan D. argues that software systems such as self-adaptive systems must embrace uncertainty within the engineering discipline of software engineering [15]. As a representative work, a method for expressing uncertainty using a partial model is proposed in [8]. A partial model can represent a specific type of uncertainty in which there are uncertain issues known and shared among the stakeholders including developers and customers. For example, there are alternative user requirements although it is uncertain which alternative should be selected. A partial model is a single model containing all possible alternative designs of a system and is encoded in propositional logic. We can check whether or not a model satisfies some interesting properties even if there are uncertain concerns.

4.6.2 Modular Programming for Uncertainty

Modularity is one of the important principles in software engineering. Unfortunately, the state-of-the-art module mechanisms do not regard an uncertain concern as a first-class software module. If uncertainty can be dealt with modularly, we can add or delete uncertain concerns to/from code whenever these concerns arise or are fixed to certain concerns.

To deal with this problem, a new programming style supporting *modularity for uncertainty* is proposed in [14]. This approach consists of three key ideas: (1) a pluggable interface for describing uncertainty, (2) interface-based modular reasoning for uncertainty, and (3) management support for tracing when and why uncertain concerns arise or are resolved. This interface called *Archface-U*, which supports *component-and-connector* architecture, consists of two kinds of interfaces, *component* and *connector*.

Figure 4.9 (Printer-scanner system), a well-known parallel system that falls into a deadlock [23], is an example of *Archface-U* descriptions. Two processes P and Q acquire the lock from each of the shared resources, the printer and the scanner, and then release the locks. The symbols $\{\}$ and $[]$ represent *alternative* and *optional*, respectively. A component is the same with ordinary Java interface. A connector, which is specified using the notation similar to FSP (finite state processes), defines the message interactions among components. FSP is based on process algebra and generates finite LTS (labelled transition systems). An arrow in FSP indicates a sequence of actions. For example, `GET` (List 1, line 22) shows that the action `scanner.get` is executed after the action `printer.get` is executed.

```
[List 1]
01: interface component cPrinter {
02:   public void get();
03:   public void put();
04:   public void print();
05:  [public void utility();]
06: }
07:
08: interface component cScanner {
09:   public void get();
10:   public void put();
11:   public void scan();
12:  [public void utility();]
13: }
14:
15: interface component cCopyMachine {
16:   public void copy();
17: }
```

```
18: interface connector cSystem (
19:   cCopyMachine P, cCopyMachine Q,
20:   cPrinter printer, cScanner scanner) {
21:
22:   GET  = (printer.get -> scanner.get);
23:   PUT  = (printer.put -> scanner.put);
24:   COPY = (scanner.scan -> printer.print);
25:
26:   P.copy = (GET -> COPY -> PUT -> P.copy);
27:   Q.copy = (GET -> COPY -> PUT -> Q.copy);
28: }
[List 2]
01: interface connector uSystem
02:   extends cSystem (
03:   cPrinter printer, cScanner scanner) {
04:
05:   GET = ({printer.get -> scanner.get,
06:           scanner.get -> printer.get});
07: }
```

Fig. 4.9 Archface-U description (Printer-scanner system)

In *Archface-U*, uncertain concerns are defined as a subinterface as shown in List 2. By extending the existing interface, we can introduce uncertainty modularly. In List 2, it is uncertain how to acquire printer and scanner resources in two processes, P and Q.

As shown in Fig. 4.9, we can explicitly represent uncertainty using *alternative* and *optional* language constructs. If a developer is writing a program and he or she becomes aware of the existence of uncertainty, the developer only has to modify *Archface-U* as shown in List 2. The developer does not have to modify the original code, because the essential information containing uncertain concerns is expressed in the *Archface-U* and the behavioural properties can be checked using only this information as explained below. If an uncertain concern is fixed to certain, a developer only has to delete the corresponding inheritance (List 2) and modify the original *Archface-U* (List 1) if needed.

4.6.3 Modular Reasoning Based on Partial Model

We can use the verification power provided by a partial model as illustrated in Fig. 4.10. A partial model is generated from *Archface-U* definitions including uncertainty represented by *alternative* and *optional*. Uncertainty is a target of compilation. The type checker verifies whether code is a subset of the partial model. From the theoretical aspect, type checking is passed when each code is a refinement of *Archface-U*. Our compiler is based on the refinement calculus focusing on simulation.

Behavioural properties represented by LTL (linear temporal logic) can be automatically verified using model checkers such as LTSA (LTS analyser) supporting FSP. If a property is verified by a model checker and the type check is successfully passed, the program satisfies important properties such as deadlock free. We show a verification process in details. In case of the printer-scanner system, there are four

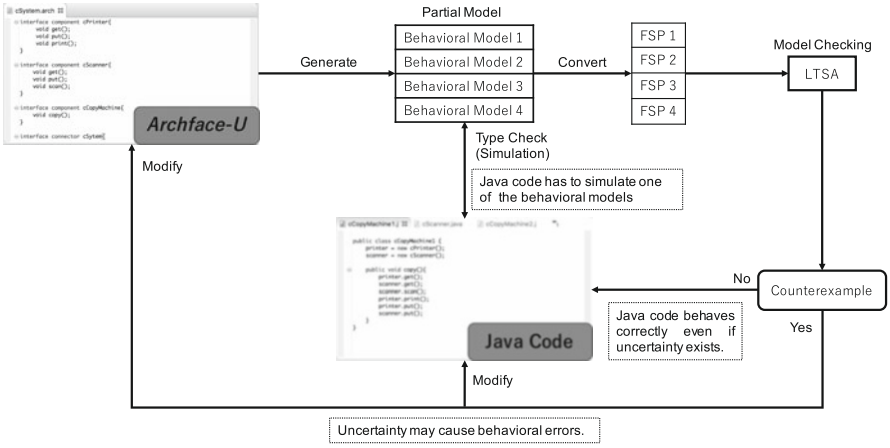

Fig. 4.10 Uncertainty-aware modular reasoning

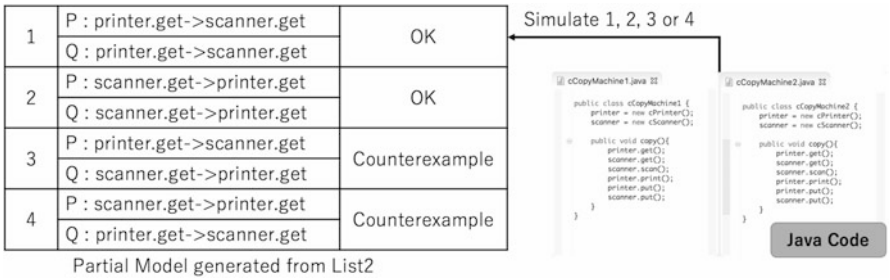

Fig. 4.11 Partial model and Java program

possible resource acquisition sequences. Type check is passed if Java code simulates one of these sequences. In Fig. 4.11, Java code simulates the sequence 1, and the type check is passed. If counterexamples are not generated by a model checker, we can select any sequence (either of 1, 2, 3, or 4 is okay). We can proceed the development even if uncertain concerns exist, because the code simulating sequence 1 is correct. Unfortunately, counterexamples are generated in case of Fig. 4.11, and these counterexamples show that the acquisition order must be the same. In this case, uncertainty may cause a deadlock although the Java code in Fig. 4.11 is correct. A developer can confirm whether or not he or she can embrace this uncertainty before modifying the code. In this case, the developer should not modify the code. As another situation, assume that a developer makes the code simulating the sequence 3 or 4. Although the type check is passed, the code is not correct because counterexamples are generated. In this case, a developer has to change the code to a new version simulating the sequence 1 or 2. In this case, a developer can resolve uncertain concerns and make a correct program before debugging and testing.

State explosion is a crucial problem when applying model checking to source code. In our approach, model checking is performed in terms of only FSP descriptions in *Archface-U*. Code is not the direct target of model checking. As a result, the number of states is reduced. Nevertheless, code can be indirectly verified by the model checker if the code conforms to its *Archface-U* via type checker. Our approach mitigates the problem of state explosion by integrating type checking with model checking.

4.6.4 Bidirectional Transformation for Uncertainty

Our approach can be regarded as an application of a bidirectional transformation. *Get* uses *Archface-U* and code to produce a partial model as a view. *Put* uses *Archface-U*, code, and a partial model to reflect changes made to the partial model into the code.

4.7 Conclusion

In this chapter, we have introduced bidirectional transformation, as well as bidirectional programming. We have shown how bidirectional programming is a technique that can be applied to various aspects of the engineering of self-adaptive systems. We targeted four areas in particular: abstraction, separation of concerns, νRule - based adaptation, and uncertainty-aware software development.

References

1. Acher, M., Collet, P., Fleurey, F., Lahire, P., Moisan, S., Rigault, J.P., et al.: Modeling context and dynamic adaptations with feature models. In: Proceedings of the 4th International Workshop on Models@run.time, Denver (2009)
2. Bohannon, A., Foster, J.N., Pierce, B.C., Pilkiewicz, A., Schmitt, A.: Boomerang: resourceful lenses for string data. In: Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '08, San Francisco, pp. 407–419. ACM, New York (2008)
3. Cheng, B.H., Lemos, R., Giese, H., Inverardi, P., Magee, J., Andersson, J., Becker, B., Bencomo, N., Brun, Y., Cukic, B., Marzo Serugendo, G., Dustdar, S., Finkelstein, A., Gacek, C., Geihs, K., Grassi, V., Karsai, G., Kienle, H.M., Kramer, J., Litoiu, M., Malek, S., Mirandola, R., Müller, H.A., Park, S., Shaw, M., Tichy, M., Tivoli, M., Weyns, D., Whittle, J.: Software engineering for self-adaptive systems. In: Software Engineering for Self-Adaptive Systems: A Research Roadmap, pp. 1–26. Springer, Berlin/Heidelberg (2009). https://doi.org/10.1007/978-3-642-02161-9_1

4. Colson, K., Dupuis, R., Montrieux, L., Hu, Z., Uchitel, S., Schobbens, P.Y.: Reusable self-adaptation through bidirectional programming. In: SEAMS'16: 11th International Symposium on Software Engineering for Adaptive and Self-Managing Systems. ACM, Austin (2016)
5. Cunha, J., Fernandes, J.P., Mendes, J., Pacheco, H., Saraiva, J.: Bidirectional transformation of model-driven spreadsheets. In: Hu, Z., de Lara, J. (eds.) Theory and Practice of Model Transformations. Lecture Notes in Computer Science, no. 7307, pp. 105–120. Springer, Berlin/Heidelberg (2012)
6. Czarnecki, K., Foster, J.N., Hu, Z., Lämmel, R., Schürr, A., Terwilliger, J.F.: Bidirectional transformations: a cross-discipline perspective. In: Paige, R.F. (ed.) Theory and Practice of Model Transformations. Lecture Notes in Computer Science, no. 5563, pp. 260–283. Springer, Berlin/Heidelberg (2009)
7. Damianou, N., Dulay, N., Lupu, E., Sloman, M.: A language for specifying security and management policies for distributed systems. Department of Computing, Imperial College, Technical Report, London (2000)
8. Famelis, M., Salay, R., Chechik, M.: Partial models: towards modeling and reasoning with uncertainty. In: 2012 34th International Conference on Software Engineering (ICSE), Zurich, pp. 573–583 (2012)
9. Fischer, S., Hu, Z., Pacheco, H.: "Putback" is the Essence of Bidirectional Programming. Technical Report GRACE-TR 2012-08, National Institute of Informatics (2012)
10. Fischer, S., Hu, Z., Pacheco, H.: The essence of bidirectional programming. Sci. China Inf. Sci. **58**(5), 1–21 (2015)
11. Foster, J.N.: Bidirectional programming languages. Ph.D. thesis, University of Pensylvania (2009)
12. Foster, J.N., Greenwald, M.B., Moore, J.T., Pierce, B.C., Schmitt, A.: Combinators for bidirectional tree transformations: a linguistic approach to the view-update problem. ACM Trans. Program. Lang. Syst. **29**(3), 17 (2007)
13. Foster, J., Pierce, B., Zdancewic, S.: Updatable security views. In: 22nd IEEE Computer Security Foundations Symposium, CSF'09, Port Jefferson, pp. 60–74 (2009)
14. Fukamachi, T., Ubayashi, N., Hosoai, S., Kamei, Y.: Conquering uncertainty in Java programming. In: Proceedings of the 37th International Conference on Software Engineering – ICSE'15, Florence, vol. 2, pp. 823–824. IEEE Press, Piscataway (2015)
15. Garlan, D.: Software engineering in an uncertain world. In: Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research, FoSER'10, Santa Fe, pp. 125–128. ACM, New York (2010)
16. Hidaka, S., Hu, Z., Inaba, K., Kato, H., Matsuda, K., Nakano, K.: Bidirectionalizing Graph Transformations. In: Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming, ICFP'10, Baltimore, pp. 205–216. ACM, New York (2010)
17. Hidaka, S., Hu, Z., Inaba, K., Kato, H., Nakano, K.: GRoundTram: an integrated framework for developing well-behaved bidirectional model transformations. In: 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE), Lawrence, pp. 480–483 (2011)
18. Hu, Z., Schürr, A., Stevens, P., Terwilliger, J.F.: Dagstuhl seminar on bidirectional transformations (BX). SIGMOD Rec. **40**(1), 35–39 (2011)
19. IBM Corp.: An architectural blueprint for autonomic computing. Technical report, 3rd edn. (2005)
20. Jin, Z.: Environment Modeling Based Requirements Engineering for Software Intensive Systems. Elsevier/Morgan Kaufmann/HZ Books, Cambridge (2018)
21. Ko, H.S., Zan, T., Hu, Z.: BiGUL: a formally verified core language for Putback-based bidirectional programming. In: Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2016, St. Petersburg, pp. 61–72. ACM, New York (2016)
22. Lanese, I., Bucchiarone, A., Montesi, F.: A framework for rule-based dynamic adaptation. In: Proceedings of the 5th International Conference on Trustworthy Global Computing, TGC'10, Munich, pp. 284–300. Springer (2010)

23. Magee, J., Kramer, J.: Concurrency: State Models & Java Programs, 2nd edn. Wiley, Hoboken (2006)
24. Montrieux, L., Hu, Z.: Towards Attribute-Based Authorisation for Bidirectional Programming, pp. 185–196. ACM, Vienna (2015)
25. Pacheco, H., Zan, T., Hu, Z.: BiFluX: a bidirectional functional update language for XML. In: 6th International Symposium on Principles and Practice of Declarative Programming (PPDP 2014), Canterbury (2014)
26. Voigtländer, J.: Bidirectionalization for free! (pearl). In: POPL 2009, Savannah, pp. 165–176. ACM (2009)
27. Weyns, D., Schmerl, B., Grassi, V., Malek, S., Mirandola, R., Prehofer, C., Wuttke, J., Andersson, J., Giese, H., Göschka, K.M.: On patterns for decentralized control in self-adaptive systems. In: de Lemos, R., Giese, H., Müller, H.A., Shaw, M. (eds.) Software Engineering for Self-Adaptive Systems II. Lecture Notes in Computer Science, no. 7475, pp. 76–107. Springer, Berlin/Heidelberg (2013)
28. Xiong, Y., Liu, D., Hu, Z., Zhao, H., Takeichi, M., Mei, H.: Towards automatic model synchronization from model transformations. In: 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), Atlanta, pp. 164–173. ACM (2007)
29. Yu, Y., Lin, Y., Hu, Z., Hidaka, S., Kato, H., Montrieux, L.: Maintaining invariant traceability through bidirectional transformations. In: 2012 34th International Conference on Software Engineering (ICSE), Zurich, pp. 540–550 (2012)
30. Zan, T., Liu, L., Ko, H.S., Hu, Z.: Brul: a putback-based bidirectional transformation library for updatable views. In: Proceedings of the 5th International Workshop on Bidirectional Transformations, Bx 2016. CEUR Workshop Proceedings, vol. 1571, pp. 77–89. CEUR-WS.org, Eindhoven (2016)

Chapter 5
Parallel Adaptation of Multiple Service Composition Instances

Rafael Roque Aschoff, Andrea Zisman, and Pedro Alexandre

Abstract Existing approaches for adaptation of service compositions do not consider the fact that common services can be used in different compositions, and, therefore, a problem that may be identified in one composition could be used to predict unwanted situations in other compositions. In this paper, we propose a parallel and proactive adaptation framework that supports proactive adaptation in multiple service composition instances at the same time. In the framework, events observed for one particular service composition instance are shared between all composition instances executed in parallel in order to better predict problems and rectify them in all necessary instances, when possible. The parallel characteristic of the framework also supports balancing the load among candidate service operations, and, therefore, it considers the maximum expected service operation throughput between the compositions. A prototype tool has been implemented to illustrate and evaluate the framework in different scenarios.

5.1 Introduction

Adaptation of service compositions is considered a major research challenge for service-based systems [6, 7, 14, 19]. Several situations may trigger the need for adaptation in service compositions, including (i) changes in or emergence of new requirements, (ii) changes in the context of the composition and participating

R. R. Aschoff (✉)
Federal Institute of Pernambuco - IFPE, Pernambuco, Brazil
e-mail: rafael.roque@palmares.ifpe.edu.br

A. Zisman
The Open University, Milton Keynes, UK
e-mail: andrea.zisman@open.ac.uk

P. Alexandre
University of Sao Paulo, São Paulo, Brazil
e-mail: pedro.alexandre@usp.br

Y. Yu et al. (eds.), *Engineering Adaptive Software Systems*,
https://doi.org/10.1007/978-981-13-2185-6_5

services, (iii) changes in functional and quality aspects of services in compositions, (iv) failures in services in compositions, and (v) emergence of new services.

More recently, some approaches to support adaptation of service composition in a reactive way [1, 4, 9, 15, 18] or proactive way [5, 22] have been proposed. However, these approaches support changes in service compositions in future executions of the composition, instead of changes in compositions during their execution. Moreover, existing approaches allow changes to be performed only in a single composition and do not consider the fact that services that need to be replaced may be participating in different compositions at the same time.

In this paper we propose a framework called ParProAdapt to support *parallel and proactive adaptation* of service compositions. The work presented in this paper extends our previous work [2, 3] that supports proactive adaptation of service composition in order to allow parallel adaptation and load balancing management of service compositions. We define *parallel adaptation* of service compositions as changes in service compositions that are being executed at the same time and which share common service operations that need to be replaced. Our approach supports the situation in which common operations are being used in different compositions, and these operations may need to be replaced in the different compositions and not necessarily only in the composition where the need for changes has been identified, for example, when the service providing the operation becomes unavailable or when the operation malfunctions. The approach also supports the situation in which several execution instances of the same composition are executed concurrently, and a problem identified in one instance also needs to be rectified in the other instances of the composition.

Another novelty of the work presented in this paper is the support for *load balancing* management. More specifically, when performing changes in a service composition, the load of a particular invoke activity can be distributed over different candidate service operations in order to increase or maintain the total throughput of the composition. If a deployed service operation is unavailable, and there are no candidate operations with the same expected throughput, a combination of more than one candidate operation can be considered.

In order to illustrate, suppose a credit card service S_{CC}, with an operation to make a payment O_{Pay}, that is used in compositions C_1, C_2, and C_3. Assume that S_{CC} becomes unavailable and this is identified when trying to invoke O_{Pay} in composition C_1. Consider that O_{Pay} has not yet been invoked during the execution of C_2 and C_3. In this case, O_{Pay} should be replaced in C_1, to allow the composition to continue its execution, as well as proactively replaced in C_2 and C_3, to avoid invoking O_{Pay} in these two compositions, and only after attempting to invoke O_{Pay}, the process realises that O_{Pay} is unavailable. The situation described above is not unrealistic since it is expected that services will be used in several applications at the same time.

In the framework the *proactive adaptation* of service compositions consists of detecting the need for changes and implementation of changes in a composition, before reaching an execution point in the composition where a problem may occur, for example, the identification that the response time of a service operation in

a composition may cause violation of the composition's service-level agreement (SLA), requiring other operations in the composition to be replaced in order to maintain the SLA, or the identification that a service provider P is unavailable requiring other services in the composition from P to be replaced, before reaching the parts in the composition where services from P are invoked.

In ParProAdapt the prediction of problems that trigger the need for adaptation is based on *function approximation* and *failure spatial correlation* techniques [16]. Moreover, the need for adaptation considers a group of operations in a composition flow, instead of isolated operations, in order to avoid replacing an operation in a composition when there is a problem, and this problem can be compensated by other operations in the composition flow.

The remainder of this paper is structured as follows. In Sect. 5.2 we present an overview of the ParProAdapt framework and provide a description of the proactive and parallel adaptation approaches used in the framework. In Sect. 5.3 we describe implementation and evaluation aspects of our work. In Sect. 5.4 we give an account of related work. Finally, in Sect. 5.5 we discuss concluding remarks and future work.

5.2 Parallel and Proactive Adaptation Framework

The main goals of the ParProAdapt framework is to provide dynamic, proactive, and parallel adaptation of service compositions. It supports a parallel identification and prediction of the need for adaptation and an autonomously reconfiguration of the service compositions during their execution time. The parallel characteristic of the approach is concerned with the identification of a problem in an instance of a composition and the impact of this problem in other instances of the same composition or in different compositions that share common services when the identified problem is in any of these services.

ParProAdapt is based on an event-based strategy in which different components of the framework generate different types of events. It supports parallel and proactive adaptation of service compositions due to four different types of situations, namely, C1, events that cause the composition to stop its execution (e.g. unavailability or malfunctioning of a deployed service operation); C2, events that allow the composition to continue to be executed, but not necessarily in its best way (e.g. the network link is congested, causing delays on the response times of some operations; such fluctuations in the response time may require adaptation in order to comply with SLA parameters of the composition); C3, emergence of new requirements (e.g. messages exchanged between services need to be encrypted; the response time of the composition needs to be improved); and C4, emergence of better services (e.g. a cheaper service becomes available).

The above situations are mapped to different events that are analysed in terms of the need for adaptation, and, depending on the results of the analysis, the adaptation process is executed. The adaptation consists of creating a valid configuration for a composition by (a) replacing a single service operation in the composition by another service operation or by a group of dynamically composed service operations

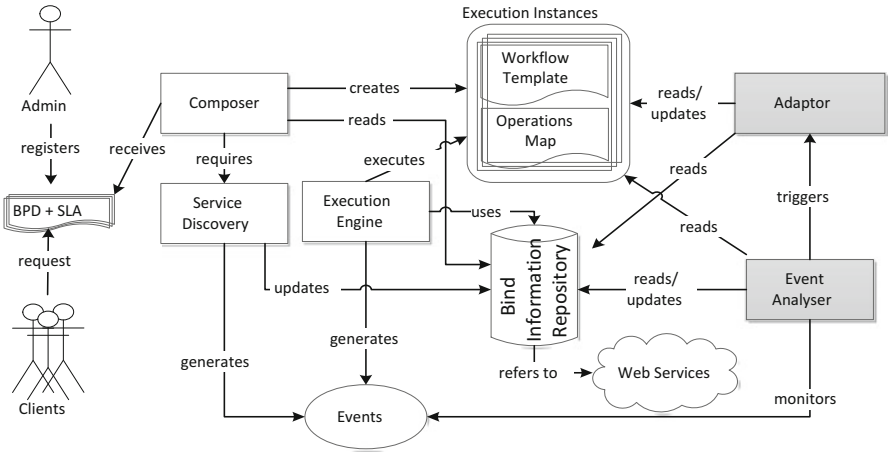

Fig. 5.1 ParProAdapt framework architecture

(replacement of types 1-1 or 1-n) or (b) replacing a group of service operations in a composition by a single operation or by a group of dynamically composed service operations (replacement of types n-1 or n-m).

The replacement of an operation may cause *signature dependency* issues with other operations in the execution instance, i.e. the situation in which the output parameter (or its part) of an operation is used as input parameter (or its part) in another operation. In the case of operation signature dependency issues, it is necessary to verify the need to replace affected operations.

The creation of a valid configuration for a composition considers the execution logic (regions) of the composition (sequence, parallel, conditional selection, and repeat) to identify a group of operations that may need to be replaced by an operation or a group of dynamically composed operations. The creation of a valid configuration is considered an optimisation problem based on the selection of appropriate combinations of candidate service operations that satisfy the SLA parameters of a composition.

Figure 5.1 shows an overview of ParProAdapt framework. The framework is composed of five mains components, namely, *composer*, *service discovery*, *execution engine*, *event analyser*, and *adaptor*, described below.

Composer This component is responsible to parse business process definitions (BPDs) (service compositions) and their associated service-level agreements (SLAs) and create an internal configuration for the service composition using the service discovery and *bind information repository*. This configuration is a *service composition execution instance*. The composer invokes the service discovery component to identify service operations that implement the logic of the service compositions and satisfy the SLA parameters of the compositions. Different configurations of a service composition execution instance may be created for the various clients requesting the composition.

Execution Instance It is composed of (i) a logic workflow of a service composition, which defines abstract operations, their order of execution and dependencies between operations and (ii) a map between the abstract operations in the workflow and the binding information for the actual services. An execution instance extends the expressiveness of a service composition with information about the (i) execution flow, (ii) deployed endpoint service operations and their locations, (iii) state of a service operation in a composition (e.g. executed, to be executed, and executing), (iv) observed QoS values of a service operation after its execution, (v) expected QoS values of a service operation, and (vii) SLA parameter values for the service operations and the composition as a whole.

Service Discovery This component identifies possible candidate service operations to be used in the composition, or to be used as replacement operations in case of problems. We assume the use of the service discovery approach that has been developed by one of the authors of this paper to assist with the identification of candidate service operations [22]. This approach advocates a proactive selection of candidate service operations based on distance measurements that match functional, behavioural, quality, and contextual aspects. The candidate service operations are identified in parallel to the execution of the compositions based on subscribed operations and are kept in a local *bind information repository*.

Bind Information Repository It keeps track of all possible service operations to be used by the service compositions, including not only the deployed operations, but also candidate service operations. The repository also contains information about the expected QoS parameters of the operations and their status (e.g. available and unavailable). The service discovery component updates the repository with information about new identified service operations or new status of already identified operations. When new operations are identified, or there are changes in the status or characteristics of existing operations, an event about the changes is generated and handled by the *event analyser* component.

Execution Engine An *execution engine* is the piece of software responsible for the execution of business processes described in the form of an executable service composition. Different service compositions can be deployed in an execution engine, and for each request of a particular composition, a *private session* must be maintained in order to individually and correctly parse input and output parameters. In the same way that a web service description (WSD) contains an abstract part for the general definitions of a web service and a concrete part for the binding information, for each service composition SC_n deployed in an execution engine, there is an abstract *composition template* T_n consisting of the workflow logic and a set of binding information for each deployed service operation S_{T_n}.

The abstract template T_n contains invoke activities pointing to abstract web service definitions. While executing a services composition SC_n, the execution engine uses the binding information S_{T_n} to identify the actual concrete operation to be invoked. Without a way to dynamically update the structural logic (T) or the binding information (S_T), compositions are bound to use the same set of

concrete operations, which results in great issues when such operations degrade their performance or present any fault.

We developed a simple execution engine that handles execution instances independently. It identifies service operations to be used and how they should be accessed. Before invoking a service operation, the execution engine requests the status of the operation and the status of the composition as a whole (e.g. when the response time for the whole composition violates the SLA parameter of the composition). In the case in which a service operation is unavailable, or there is not a match between the expected and observed QoS values of an operation, a new event is created and sent to the event analyser.

Event Analyser This component is responsible for analysing all the generated events in order to predict unwanted situations and execute parallel changes in the execution instances, when necessary. More details of the functionality of this component are discussed in the subsections below.

Adaptor This component is responsible to execute individual changes in the execution instances, based on requests received from the event analyser. In order to execute the necessary changes, the adaptor component reads information from the bind repository about available operations.

5.2.1 Proactive Adaptation Approach

The proactive adaptation approach used by the framework has been described in details in [2, 3]. In this section we provide an overview of the approach for completeness of this paper and to better understand the parallel characteristics of the approach, which is the novel aspect of the paper.

As described above, the adaptation process may be triggered by situations of types C1 to C4. The events generated for situations C1 to C3 may produce unwanted situations resulting in failure in the execution of service compositions. Due to the nature of situations, C3 and C4, they cannot be predicted. However, prediction techniques can be used to support situations C1 and C2. For any of the situations that may trigger the need for adaptation, the process tries to identify other parts in the execution instance that may be affected by the situation. The process is based on the use of two techniques executed by the event analyser component, namely, (a) QoS analysis and (b) spatial correlation analysis.

QoS Analysis It consists of a failure prediction technique that verifies the impact that changes of QoS values of deployed service operations may have in the SLA parameters of a composition as a whole. This analysis is used to avoid replacing an operation in an execution instance when the problem can be compensated by other operations in the execution flow. The process also identifies other operations in the instance that may be affected due to violation of QoS values.

In the framework, the process concentrates on the analysis of violations of response times and cost values of the operations. The analysis is based on the use of exponentially weighted moving average (EWMA) [13] for modelling the expected service operation QoS values. An expected QoS value (e.g. response time) of an operation is calculated based on previous observed QoS values for that operation. The new expected QoS value of an operation is updated on the bind information repository. The aggregated QoS values of an execution instance is calculated based on the expected QoS values of the operations not yet executed, and the observed QoS values of the operations are already executed. The computation of the aggregated QoS values for the whole composition depends on the type of the QoS values and the logic workflow structures of the composition (e.g. conditional, sequence, parallel, and repeat logic structures).

When there is no violation of the SLA values for the whole composition, there is no need to adapt the execution instance. If the expected values are violated, the adaptor component is invoked to identify a valid configuration for the composition. This valid configuration may be generated by replacing operations in the execution instance that have not yet been executed and by attempting to find possible combinations of replacement operations that provide the functionality of those operations and maintain the SLA values of the composition.

Spatial Correlation Analysis This technique consists of identifying spatial correlations between operations, services, and providers. It is concerned with the situation in which providers, services, and operations become unavailable and the impact that this unavailability may have in other services or operations being used in the composition. For example, consider a service S that becomes unavailable. In this case, the process considers all other operations of S in the composition since these operations may not be able to be executed. Similarly, when a provider P is unavailable, all services and operations provided by P are also marked as out of reach on the bind information repository.

During the spatial correlation analysis, the bind information repository is updated about the availability of operations. In the case in which operations deployed in the execution instances are identified as unavailable, the adaptor is invoked to identify a valid configuration for the composition. In the spatial correlation analysis, all running execution instances are aware of any issues when trying to invoke operations with a problem.

When using only the proactive adaptation approach, for the trigger situations C1 and C2, the running execution instances identify a problem with an operation only when they reach a point of execution in which they request the operation with the problem. The other parts of the execution instances that may be affected by the operation with a problem will be proactively identified based on the techniques discussed above. However, to allow running execution instances to be notified about a problem in a deployed operation, as soon as possible, for any of the trigger situations, we propose the parallel adaptation approach described below.

5.2.2 *Parallel Adaptation Approach*

The parallel adaptation approach complements the proactive adaptation approach with two new techniques, namely, (a) parallel analysis and (b) load balancing analysis. Overall, the idea of the parallel approach is to identify other running execution instances that may be affected by a problem identified in one execution instance and rectify this problem in these other instances in parallel, during their execution time. As mentioned before, those execution instances can be copies of the same service composition in which a problem was identified or different service compositions that use an operation for which there is a problem.

Parallel Analysis With this technique, it is possible to reduce the time that it is necessary to identify a problem in an operation used in a service composition, the assessment of this problem in other running service compositions that share the operation and the execution of the actions to rectify the problem.

Our approach allows instances of service compositions to be adapted in parallel independent of each other. More specifically, changes executed in one service composition instance do not necessarily interfere with other instances which are executed in parallel, even when these are instances of the same service composition.

Figure 5.2 presents a snapshot of the above characteristics of our approach. As shown in the figure, for each request m of a deployed service composition SC_n, an execution instance EI_m^n is created using the composition template T_n and its respective binding information S_{T_n}. The framework creates for each execution instance EI_m^n a private template T_m^n and binding information for this template $S_{T_m}^n$

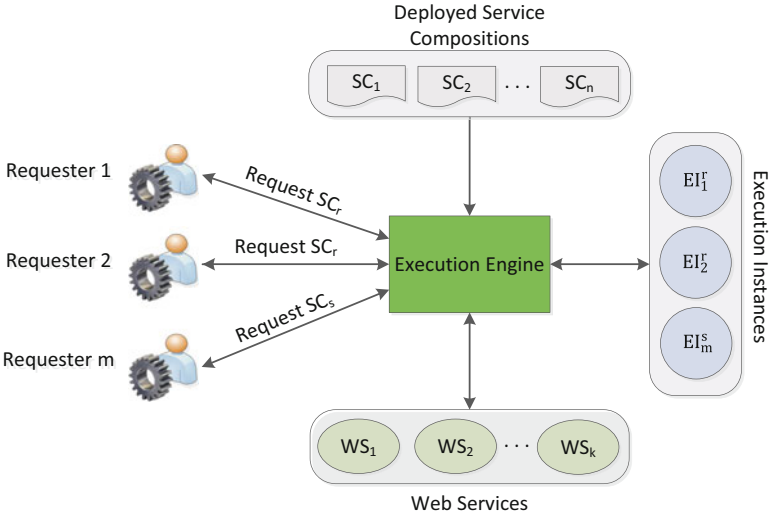

Fig. 5.2 Illustration of the execution engine accessing execution instances of service composition

in order to allow adaptations of a particular service composition instance. The approach also maintains the composition template T_n and its binding information S_{T_n} to support proactive adaptation of new instances of a service composition that may be created due to future requests. In order to illustrate, consider three execution instances EI_1^r, EI_2^r, and EI_1^s for service compositions SC_r and SC_s respectively. Changes in the private template T_1^r of EI_1^r or changes in its binding information $S_{T_1}^r$ do not create direct changes in the private templates T_2^r and T_1^s or in the binding information $S_{T_2}^r$ and $S_{T_1}^s$. However, it is possible to allow the adaptation across parallel execution instances of the same or different service compositions by accessing their private templates and binding information. Moreover, given a set of deployed service compositions $\{SC_1, SC_2, \ldots, SC_n\}$, new execution instances of these compositions can benefit from previous processed information by changing the respective composition templates $T_x, 1 \leq x \leq n$ or the respective default binding information $S_{T_x}, 1 \leq x \leq n$, which are both used to create the new execution instances.

As described in Sect. 5.2.1, in the proactive approach, the verification of the status of deployed operations is only executed when a running execution instance reaches a point of execution in which a deployed operation is requested. With the parallel analysis, the event analyser component triggers parallel adaptation of all affected execution instances. This allows parallel execution instances to reconfigure themselves earlier in the running process (before reaching the operation with issues) and, therefore, augment the probability of success in the adaptation since there will have potentially more options for changes in the composition, for example, in the case in which it is necessary to change a group of operations in the composition that have not yet been executed, in order to conform to the SLA values of the composition.

The parallel analysis is executed by verifying if each running execution instance has a valid configuration, before the execution of each deployed operation in the instances. The verification of a valid configuration consists of analysing if there is any operation not yet invoked that may have become unavailable and if there are any SLA violations due to QoS discrepancies. This verification is executed by the event analyser based on the information in the execution instances and the bind information repository (see Fig. 5.1). During the above verifications, an execution instance cannot proceed with its execution until either an adaptation is performed or it is concluded that there is no need for adaptation.

Load Balancing This technique is used to verify if the throughput of the service compositions are maintained as initially specified for the compositions. The throughput specified for a service composition is reflected in the activities and their deployed operations in the composition. The throughput of each service operation in an execution instance is calculated and compared with the maximum accepted throughput value of the composition, in order to avoid overloading the use of the deployed operations. This is done by using a throughput counter for each deployed operation. When an execution instance is created, the counters associated with the operations are incremented; when the operations are invoked during the execution

of an instance, their associated counters are decremented. The maximum accepted throughput value of an operation is maintained in the bind information repository to allow the composer and adaptor components know which operations can be used in an execution instance, without causing operation overload.

In the case in which a deployed operation O needs to be replaced, the approach supports the use of one or more operations to replace O when these operations provide the same functionality of O and the sum of the throughput values of these operations are equal to the throughput value specified for the activity associated with O. The above is possible due to the parallel approach being described in the paper since the framework keeps track of the parallel use of all operations in the execution instances that are running at the same time.

5.3 Implementation and Evaluation

In order to demonstrate and evaluate the work, we have implemented a prototype tool of the framework in Java. The tool assumes service compositions in WS-BPEL [20] exposed as web services using SOAP protocol, and participating operations and user requests emulated using SoapUI. The service discovery tool was also implemented in Java and is exposed as a web service.

In our previous paper [3], we showed how computationally inexpensive and scalable are the various activities concerned with the proactive adaptation aspect of the framework for a single service composition. In particular, we analysed the time to identify and resolve SLA violations, the time to identify and resolve signature dependencies, the time to identify spatial correlations, and the time to adapt a composition by changing groups of operations in a composition. In the current parallel approach, the activity that generates additional computational effort is concerned with the reconfiguration algorithm of a service composition and its additional analysis of the load of operations. Therefore, we consider that the parallel extension is aligned with our previous results with respect to the computation of these activities. In this paper, our focus is to demonstrate if there are improvements in the adaptation process when considering the parallel adaptation, in terms of the number of service compositions that can adapt successfully. In other words, we are trying to verify if our approach is able to improve the dependability of service compositions by dealing with two specific types of problems, namely, (a) a problem that can only be solved by changing the identified faulty operation and a set of other operations which are logically presented in the compositions prior to the faulty one and (b) a problem where there is no single operation that can support the number of requests being generated.

As previously discussed, our approach is able to adapt parallel execution instances of single or multiple service compositions. The adaptation process itself, however, makes no difference if the execution instances are of the same or distinct compositions. Such distinction exists only during the initial phase when the execution instance must be created based on the template of a deployed composition.

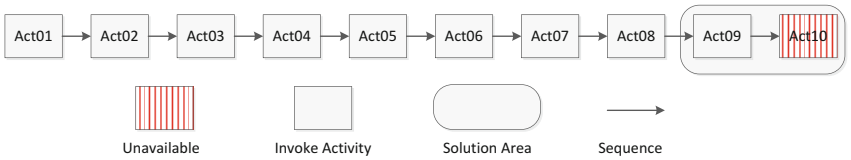

Fig. 5.3 Service composition workflow for evaluation of Scenario 1

In order to make the evaluation and discussion of the results more clear, we decided to conduct our experiments with multiple instances of the same service composition.

In the experiments, we assume that each of the various execution instances starts its execution in different time steps. We also consider that the number of running execution instances at different points of their execution flow is approximately the same. More specifically, the number of running instances executing the initial part of their flows is similar to the number of those in the middle or in the end of their execution flows. The work has been evaluated for three main cases with different scenarios. In the first two cases, we compare the use of the parallel and proactive approach with the proactive approach only for a service composition with a linear structure (Case 1) and a complex service composition (Case 2). For both cases (Case 1 and Case 2), we assume that one or more operations in the composition become unavailable. However, the approach supports a similar process for the other types of problems (e.g. violation of QoS values of an operation). Finally, in Case 3 we evaluate the load balancing technique.

Case 1 – Scenario 1: In this scenario, we use a service composition with a sequential workflow formed by ten *invoke activities*, as shown in Fig. 5.3. We assume that at a certain time in the experiment, the service operation assigned to the last invoke activity (Act10) becomes unavailable. Consider the existence of a set of candidate service operations for each invoke activity (Act1–Act10) presented in Fig. 5.3, and the use of any of the available candidate service operations for Act10, along with the current assigned operations for (Act1–Act09), would cause a violation of the SLA value of the whole composition. Consider the existence of a valid configuration for the service composition when replacing both the operations assigned for activities Act9 and Act10.

We compared a number of execution instances that (a) were able to adapt successfully (successful), (b) were not able to adapt (unsuccessful) and (c) did not require adaptation because they were not affected by the problem (not required), for the case in which we used the parallel and proactive approach with the case in which we used only the proactive approach. We considered 50, 100, 150, and 200 execution instances of the composition shown in Fig. 5.3.

Figure 5.4 presents the results of this experiment. For each different number of execution instances considered in the experiment, the first column represents the results when using the parallel and proactive approaches (specified as *parallel* for simplicity), while the second column represents the results when using only the proactive approach (specified as *proactive* for simplicity).

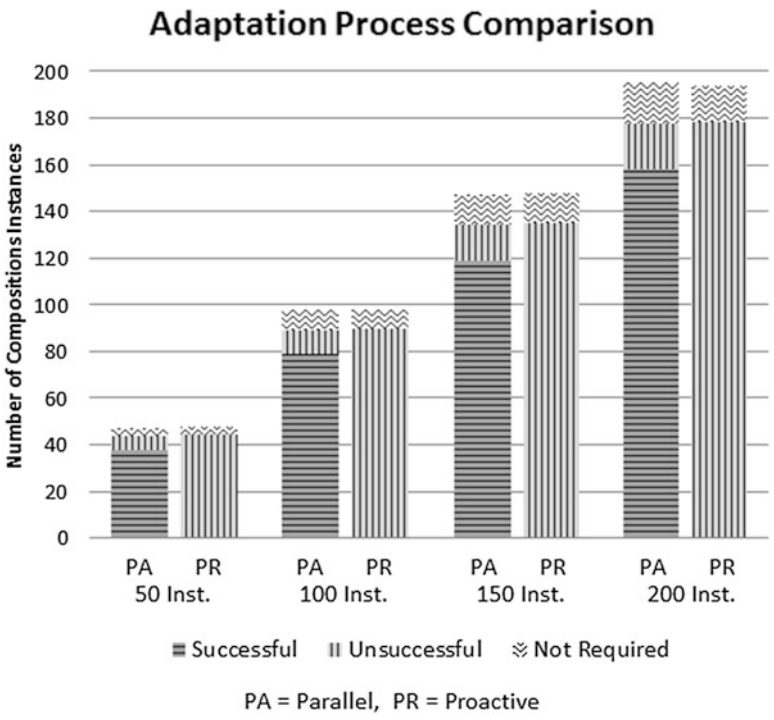

Fig. 5.4 Comparison of the adaptation process for Case 1 – Scenario 1.

As shown in Fig. 5.4, when using the combined parallel and proactive approaches, there are many more instances that are adapted and finished successfully. This is because in the parallel approach, several execution instances that are still in operation are notified about the unavailability of the operation associated with Act10 and have not yet executed the operation associated with Act09. Contrary, in the case when only the proactive approach is used, the adaptation process is attempted when the execution process tries to invoke the operation associated with Act10 and realises that this operation is unavailable. In this scenario, the process requires the replacement of the operation associated with Act09 as well. However, when attempting to invoke the operation associated with Act10, the operation for Act09 has already been executed and cannot be replaced.

Figure 5.4 also shows that even when using the proactive approach only, some instances are able to finish successfully for all the different numbers of execution instances used in the experiment. These instances are the ones that managed to invoke the operation associated with Act10 before this operation became unavailable and, therefore, were able to finish their execution successfully.

Case 1 – Scenario 2: In this scenario, we use the same service composition of Scenario 1, but we consider different positions in the composition where the operation associated with an activity becomes unavailable. We consider the

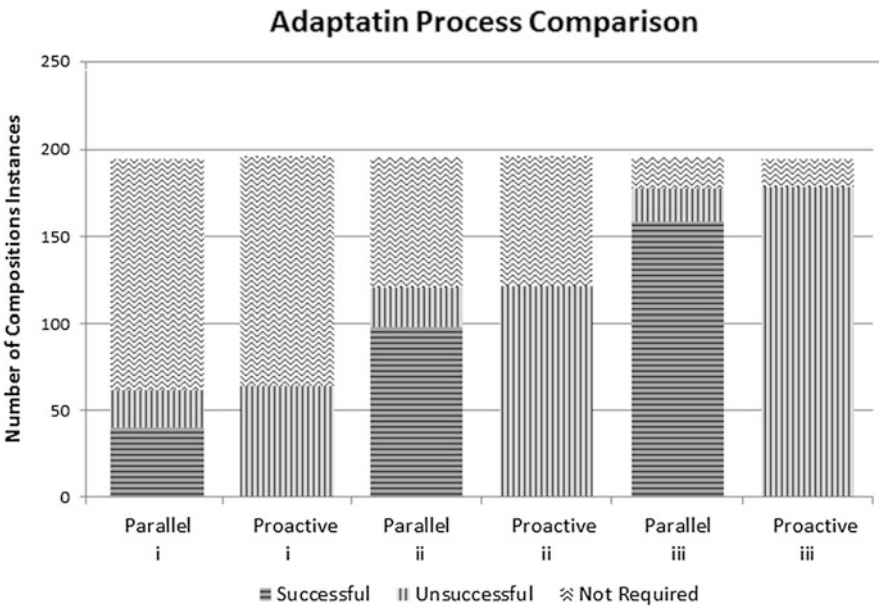

Fig. 5.5 Comparison of the adaptation process for Case 1 – Scenario 2

situations in which the operations associated with activities Act04, Act07, and Act10 become unavailable at the same time. In all three cases, we assume that a valid configuration exists when replacing both the operation that becomes unavailable and the ones associated with the previous activity of the unavailable operations, i.e. operations associated with (i) Act03 and Act04, (ii) Act06 and Act07 and (iii) Act09 and Act10. We assume 200 instances of the service composition executed at the same time.

Figure 5.5 shows the results of the experiments for situations (i) to (iii) above. As shown in the figure, when using only the proactive approach, in any of situations (i) to (iii), none of the execution instances could be successfully adapted. This is because the execution instances have already invoked the operations associated with the activities that occur before the activities that become unavailable (activities Act03, Act06, and Act09).

The results also show that the number of execution instances that do not require adaptation decreases when the problem occurs at a position closer to the end of the composition.

The number of unsuccessful adaptation instances, however, increases. This is due to the number of running execution instances that are at a point *before*, *the same*, or *after* the point in which the problem is identified, during their execution. This also explains the reason for having similar numbers of execution instances that do not require adaptation, for the parallel and proactive approach and the proactive approach only, in situations (i) to (iii).

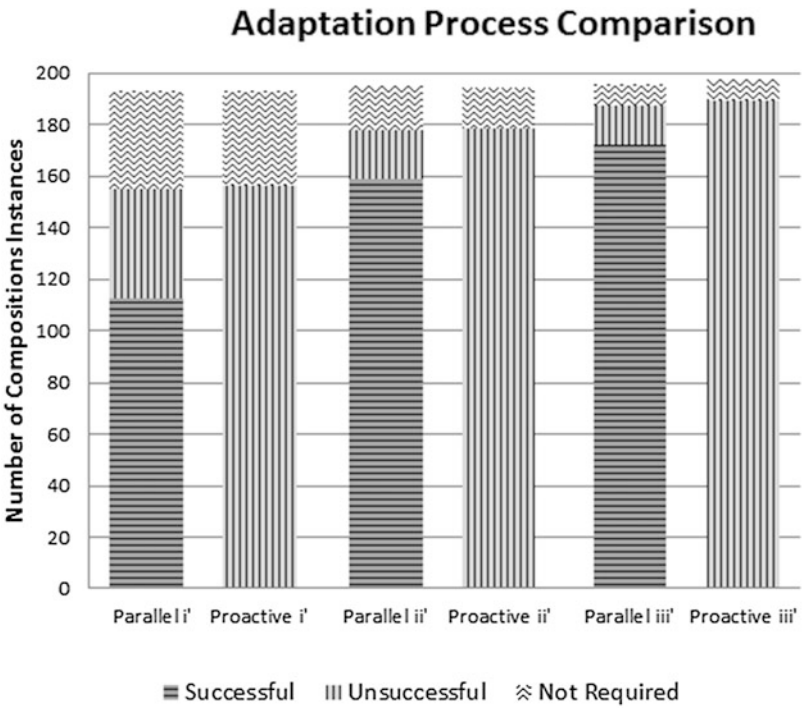

Fig. 5.6 Comparison of the adaptation process for Case 1 – Scenario 3

From Fig. 5.5 we observe that when using the parallel and proactive approaches, the number of successful adaptation instances increases, as the problem occurs at a position closer to the end of the composition. This is because the number of execution instances that can be adapted increases, since there are more instances at execution points before the operation becomes unavailable.

Case 1 – Scenario 3: In this scenario, we compare the approaches when using service compositions with a sequential structure, as in Scenarios 1 and 2, but of different sizes. We considered compositions with (i') 5, (ii') 10 and (iii') 15 activities. Similar to the above scenarios, we assume that in each of the three compositions, the operation associated with the last activity becomes unavailable and that a valid configuration exists when replacing both the operation that becomes unavailable and the operation associated with the previous activity. We assume 200 instances of the service composition executed at the same time.

Figure 5.6 shows the results of the experiments for compositions (i') to (iii'). The results in the figure show an increase in the number of execution instances that required adaptation as the size of the compositions increase. As in the case of Scenario 2, this is due to the number of running execution instances that are at a point before, the same, or after the point in which the problem is identified, during their executions. Similarly, the results show an increase in the number of successful

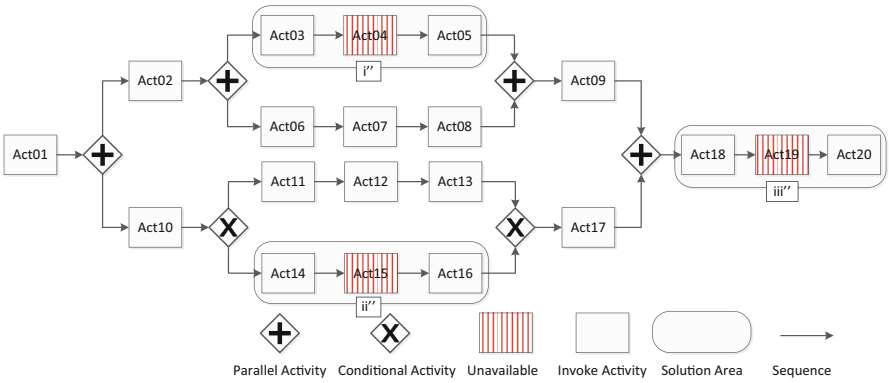

Fig. 5.7 Complex service composition workflow

adaptations for bigger compositions. Similar to Scenarios 1 and 2, the results show that when using only the proactive approach, in any of situations (i') to (iii'), none of the execution instances could be successfully adapted.

Case 2: In this case we use the service composition shown in Fig. 5.7. We consider that the operations associated with activities Act04, Act15, and Act19 become unavailable. We also assume that for each unavailable operation, the solution of a valid configuration exists when replacing the operations associated with the previous and next activities of the unavailable operation and the operation that becomes unavailable. We assume 100 instances of the service composition executed at the same time.

The example in Case 2 differs from the scenarios in Case 1 since (a) the service composition is more complex with more activities organised in different execution logics (conditional and parallel), (b) a valid configuration for the composition includes the replacement of operations associated with activities before and after the operation that becomes unavailable and (c) the operations that become unavailable are associated with activities in different execution logics.

The results of this experiment are shown in Fig. 5.8 for the unavailability of (i") Act04, (ii") Act15, and (iii") Act19. As it was expected, when the operation that becomes unavailable is at the end of the composition (situation (iii")), a larger number of execution instances require adaptation since there are more running instances at execution points before the operation becomes unavailable (as in the previous scenarios). The results show that for situation (i"), half of the execution instances did not require adaptation. For those execution instances that required adaptation, half of them were successfully adapted.

We also observe that situation (i") required more instances to be adapted than situation (ii"). This is due to the fact that situation (ii") is a conditional execution logic, and, therefore, not necessarily all the execution instances will execute this path in the composition. This is not the case in situation (i") in which all the instances need to execute the respective path in the composition.

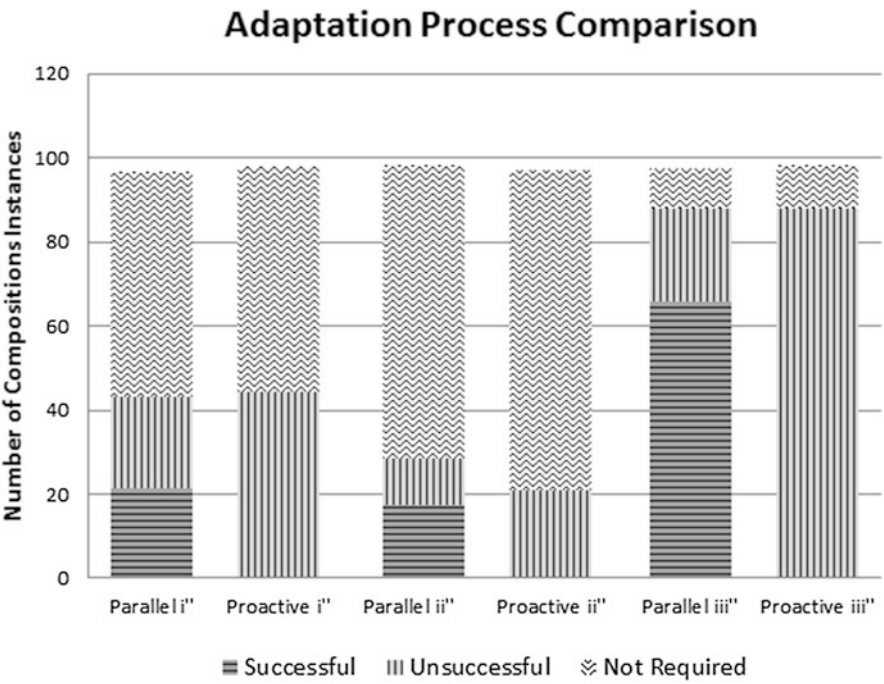

Fig. 5.8 Comparison of the adaptation process for Case 2

Case 3: In our approach, the efficiency of the proposed technique to dynamically distribute the load of service operation requests among different service providers, and in parallel with the execution of service composition instances, depends on the number of requests for a particular service operation and the capacity of the service operation to fulfil its requests. The size, complexity and logic of a service composition do not cause impact to the load balancing technique. Therefore, in order to evaluate the load balancing technique, we used a simple service composition. More specifically, the evaluation was executed in a scenario with a single invoke activity (IA) deployed in two operations given by two different providers P_1 as OP_1 and P_2 as OP_2. We assumed both OP_1 and OP_2 configured with a processing time of 1 s. Moreover, in the experiment we used a maximum of 20 concurrent service composition instances and configured OP_1 and OP_2 to be able to handle up to ten concurrent requests.

In order to introduce some random behaviour in the income rate of operation requests, we simulated the compositions requests and assumed that each request respects a uniform distribution with minimum zero and maximum one. In other words, each of the 20 parallel processes generating concurrent requests sleeps for a specific amount of time and generates a new request. After that, the process starts again if the experiment is not over. This behaviour is depicted in Fig. 5.9.

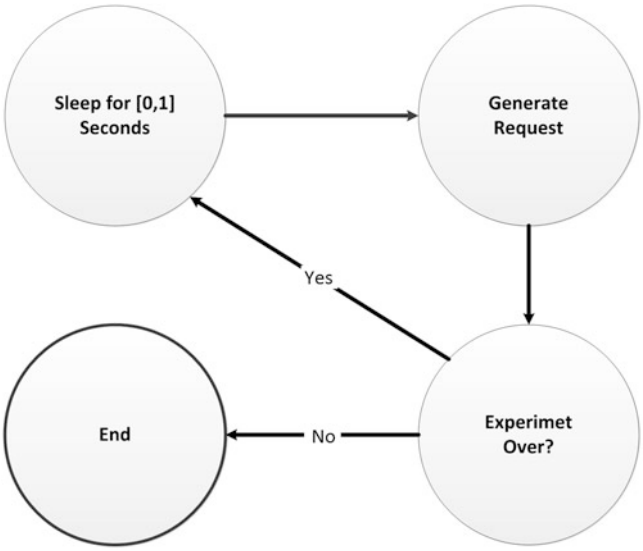

Fig. 5.9 State machine of the concurrent requests generator process

In the above-described experiment, we expected to observe an improvement in the overall performance of the execution engine in terms of the number of successfully concluded composition requests. The basic idea is that if no distribution of the load is in place, the best thing that an approach can do is to jump from one operation to another as soon as it is detected as unavailable (e.g. an operation that is not responding due to high traffic). Moreover, considering that no single operation is suitable to answer all concurrent instances, it is almost mandatory to employ some form of online testing to discover if the operation becomes available again. Without a way to assess the availability of an operation, the composition instances would just fail to be created since the system would indicate that no operation is available to perform the required tasks. We implemented a basic online testing procedure that periodically checks if the previously failed operation is available and marks it in the local repository as available again.

In our experiments, using the parallel adaptation with load balancing techniques, the average time to finish an execution instance was about 1 s. This was expected since the processing time of both OP_1 and OP_2 is configured as 1 s. Moreover, there was only at most 20 concurrent requests, and the combined throughput for OP_1 and OP_2 was 20. Therefore, no extra issues were introduced. We noted that in the case in which the ability to distribute the load between different operations was turned off, the average time to conclude an execution instance rose to about 3 s. This was due to the fact that now the adaptor component had to constantly face an error due to the high load of requests in either OP_1 or OP_2. Figure 5.10 presents a snapshot of the distribution of the requests made to OP_1 in both experiments. As we can see, when the load balancing technique is in place, the load distribution is much more

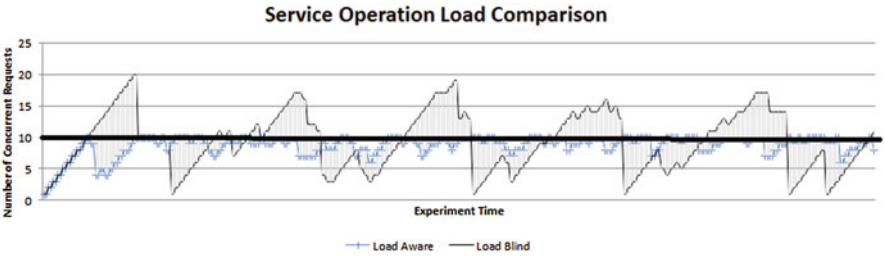

Fig. 5.10 Comparisons of the distribution of operation request for a single provider between the approach with and without load balancing

homogeneous. The black line at ten concurrent requests indicates the threshold for the capability of OP_1. Given that the approach with load balancing respects the threshold for individual operations by identifying its maximum capability, no issues are introduced. However, when such awareness is removed, and there is no way to alleviate the load, the threshold is not respected. This causes errors and requires adaptations to be executed.

5.4 Related Work

The work presented in this paper is concerned with approaches that support dynamic adaptation of service compositions, which is considered a major research challenge for service-based systems [6, 7, 14]. Initial approaches were proposed to support adaptation of service compositions in a reactive way [1, 4, 9, 15]. These approaches support adaptation of service composition based on predefined policies [4], self-healing of compositions based on detection of exceptions and repair using handlers [15], context-based adaptation of compositions using negotiation and repair actions [1] and key performance indicator analysis [9].

Other approaches have recently been proposed to support adaptation of service compositions in a proactive way [2, 3, 5, 10, 11, 19]. The work by Dai et al. [5] uses semi-Markov models for performance predictions, service reliability model, and minimization in the number of service reselection in case of changes. The decision to adapt is based on the performance of a single service. One of the first works to use a proactive approach is PREvent [10], which was designed to support prediction and prevention of SLA violations in service compositions based on event monitoring and machine learning techniques. The works by Metzer et al. [11] and Tosi et al. [19] advocate the use of testing to anticipate problems in service compositions and trigger adaptation requests. However, the creation of test cases is not an easy task.

Approaches to support multilayered monitoring and adaptation of service compositions have been proposed [8, 17, 21]. Some of these approaches use the concepts of adaptation taxonomy and templates (patterns) created during design time to

represent possible solutions for adaptation problems [17]. Other approaches rely on dynamic identification of cross-layered adaptation strategies for software and infrastructure layers [8, 21] or on the use of aspect-oriented techniques to support adaptation of compositions due to QoS aspects [12].

Our framework differs from the above approaches since it supports parallel adaptation of running execution instances. In addition, it allows for parallel and proactive adaptation of service compositions due to different types of problems and provides different ways of adapting the compositions.

5.5 Conclusions and Future Work

In this paper we described ParProAdapt framework, a parallel and proactive adaptation framework that supports parallel identification and prediction of the need for adaptation and reconfiguration of the service compositions during their execution time. The framework supports the identification of a problem in an instance of a composition and the impact of this problem in other instances of the same composition or in different compositions that share common operations when the identified problem is in any of these operations. When a problem is identified in an instance of a composition, other affected parts of the composition are proactively identified, in order to rectify the various composition instances. A prototype tool has been implemented, and the approach has been evaluated in several scenarios. The results of the evaluation demonstrate that the use of a proactive approach combined with a parallel approach outperforms the use of only a proactive approach in terms of the number of composition instances that are successfully adapted.

Currently, we are extending the framework to support service compositions that provide interactions with users and how the proactive and parallel adaptation can deal with these interactions and delays that may be caused by them. We are also investigating the use of the congestion control algorithm used in the TCP protocol to dynamically adjust the expected throughput of service operations. Another future work consists of considering different types of constraints when attempting to adapt a service composition (e.g. stateful services and operations that need to be used by certain service providers).

References

1. Ardagna, D., Comuzzi, M., Mussi, E., Pernici, B., Plebani, P.: PAWS: A framework for executing adaptive web-service processes. IEEE Softw. **24**(6), 39–46 (2007)
2. Aschoff, R., Zisman, A.: QoS-driven proactive adaptation of service composition. In: ICSOC'11, pp. 421–435 (2011)
3. Aschoff, R., Zisman, A.: Proactive adaptation of service composition. In: SEAMS'12, pp. 1–10 (2012)

4. Baresi, L., Di Nitto, E., Ghezzi, C., Guinea, S.: A framework for the deployment of adaptable web service compositions. SOCA **1**(1), 75–91 (2007)
5. Dai, Y., Yang, L., Zhang, B.: QoS-driven self-healing web service composition based on performance prediction. J. Comput. Sci. Technol. **24**(2), 250–261 (2009)
6. Di Nitto, E., Ghezzi, C., Metzger, A., Papazoglou, M., Pohl, K.: A journey to highly dynamic, self-adaptive service-based applications. ASE **15**(3), 313–341 (2008)
7. Dustdar, S., Papazoglou, M.P.: Services and service composition – an introduction (services und service komposition – eine einführung). Inf. Technol. **50**(2), 86–92 (2009)
8. Guinea, S., Kecskemeti, G., Marconi, A., Wetzstein, B.: Multi-layered monitoring and adaptation. In: ICSOC'11 (2011). https://doi.org/10.1007/978-3-642-25535-9_24
9. Kazhamiakin, R., Wetzstein, B., Karastoyanova, D., Pistore, M., Leymann, F.: Adaptation of service-based applications based on process quality factor analysis. In: LNCS'09 (2009)
10. Leitner, P., Michlmayr, A., Rosenberg, F., Dustdar, S.: Monitoring, prediction and prevention of SLA violations in composite services. In: ICWS'10 (2010)
11. Metzger, A., Sammodi, O., Pohl, K., Rzepka, M.: Towards pro-active adaptation with confidence: augmenting service monitoring with online testing. In: SEAMS'10 (2010). http://doi.acm.org/10.1145/1808984.1808987
12. Moser, O., Rosenberg, F., Dustdar, S.: Non-intrusive monitoring and service adaptation for WS-BPEL. In: WWW'08 (2008). https://doi.org/10.1145/1367497.1367607
13. Natrella, M.: e-Handbook of Statistical Methods. Nist/Sematech (2010). http://www.itl.nist.gov/div898/handbook/
14. Papazoglou, M.P., Traverso, P., Dustdar, S., Leymann, F.: Service-oriented computing: a research roadmap. Int. J. Coop. Inf. Syst. **17**(2), 223–255 (2008)
15. Pernici, B.: Self-healing systems and web services: the WS-DIAMOND approach. In: LNBIP'09 (2009)
16. Pistore, M., Marconi, A., Bertoli, P., Traverso, P.: Automated composition of web services by planning at the knowledge level. In: IJCAI'05 (2005)
17. Popescu, R., Staikopoulos, A., Liu, P., Brogi, A., Clarke, S.: Taxonomy-driven adaptation of multi-layer applications using templates. In: SASO'10 (2010). https://doi.org/10.1109/SASO.2010.23
18. Saboohi, H., Amini, A., Herawan, T., Kareem, S.: Failure recovery of composite semantic services using expiration times. In: Herawan, T., Deris, M.M., Abawajy, J. (eds.) Proceedings of the First International Conference on Advanced Data and Information Engineering (DaEng-2013), Lecture Notes in Electrical Engineering, vol. 285, pp. 683–690. Springer, Singapore (2014). https://doi.org/10.1007/978-981-4585-18-7_77
19. Tosi, D., Denaro, G., Pezze, M.: Towards autonomic service-oriented applications. Int. J. Autom. Comput. **1**, 58–80 (2009). https://doi.org/10.1504/IJAC.2009.024500
20. Web Services Business Process Execution Language (WS-BPEL) Version 2.0.: Organization for the Advancement of Structured Information Standards (OASIS) (2007). http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html
21. Zengin, A., Kazhamiakin, R., Pistore, M.: Clam: cross-layer management of adaptation decisions for service-based applications. In: ICWS'11 (2011). https://doi.org/10.1109/ICWS.2011.76
22. Zisman, A., Spanoudakis, G., Dooley, J., Siveroni, I.: Proactive and reactive runtime service discovery: A framework and its evaluation. IEEE Trans. Softw. Eng. **39**(7), 954–974 (2013)

Chapter 6
Assessing Security and Privacy Behavioural Risks for Self-Protection Systems

Yijun Yu, Yoshioka Nobukazu, and Tetsuo Tamai

Abstract Security and privacy can often be considered from two perspectives. The first perspective is that of the attacker who seeks to exploit vulnerabilities of the system to harm assets such as the software system itself or its users. The second perspective is that of the defender who seeks to protect the assets by minimising the likelihood of attacks on those assets. This chapter focuses on analysing security and privacy risks from these two perspectives considering both the software system and its uncertain environment including uncertain human behaviours. These risks are dynamically changing at runtime, making them even harder to analyse. To compute the range of these risks, we highlight how to alternate between the attacker and the defender perspectives as part of an iterative process. We then quantify the risk assessment as part of adaptive security and privacy mechanisms complementing the logic reasoning of qualitative risks in argumentation (Yu et al., J Syst Softw 106:102–116, 2015). We illustrate the proposed approach through the risk analysis of examples in security and privacy.

6.1 Introduction

Security properties are often described using confidentiality, integrity, availability, authentication and authorisation according to the ISO/IEC 9126 standard [5], which highlight the protection of asset values from malicious attacks. Privacy properties, as understood by "the rights to be left alone" [13], concern the control of sharing

Y. Yu (✉)
The Open University, Milton Keynes, UK
e-mail: y.yu@open.ac.uk

Y. Nobukazu
National Institute of Informatics, Tokyo, Japan
e-mail: nobukazu@nii.ac.jp

T. Tamai
Hosei University, Tokyo, Japan
e-mail: tamai@hosei.ac.jp

Y. Yu et al. (eds.), *Engineering Adaptive Software Systems*,
https://doi.org/10.1007/978-981-13-2185-6_6

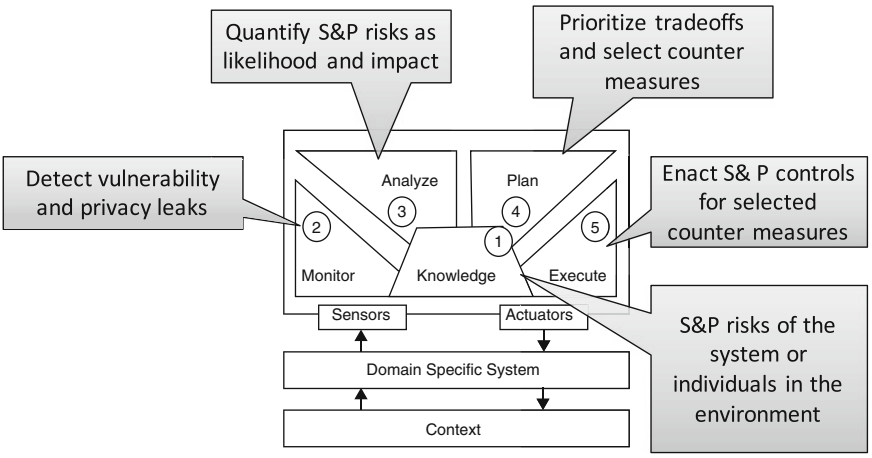

Fig. 6.1 Security and privacy concerns on the MAPE-K architecture [4] for self-protection

the identity of individuals or groups to prevent potential harms to their life and can be represented using selective disclosure [12], contextual integrity [1], etc.

Both security and privacy properties are adaptive in nature. From the dimensions of self-adaptive systems, known as MAPE-K feedback loops [2], both security and privacy can be seen as cross-cutting concerns to all these five dimensions at runtime. Self-adaptive security and privacy mechanisms, or *self-protection*, instantiate the MAPE-K dimensions [4] as follows (see Fig. 6.1).

Monitoring aims to detect system vulnerability and privacy leaks. *Analysis* requires a quantification of risks in terms of assessing of the likelihood and impact of runtime incidents. *Planning* involves the ranking, prioritisation and trade-offs of incident responses in order to select the best countermeasures at runtime. *Execution* enacts the defence to control the managed system or individuals and implement the countermeasures. Throughout these activities, the *knowledge* about the system and the individuals also change over time, which could change the predefined boundaries between attackers and defenders.

According to our earlier studies on security risks [16] and privacy arguments [12], it is necessary to analyse contextual factors in order to identify and assess the risk factors. These approaches proposed to use problem-oriented analysis on the context of a system in its running environment, in order to elicit the risk factors from a prepared knowledge base (e.g. common vulnerability exposures[1] and common vulnerability scoring system[2]).

In addition to MAPE-K feedback loops for self-protection, system and individuals also need to quantify the security and privacy risk factors at runtime. Such quantified risks could help tune the *set points* [3] for runtime security and privacy feedback loop controls. However, runtime properties of the system and environment

[1] https://cve.mitre.org

[2] https://nvd.nist.gov/vuln-metrics/cvss

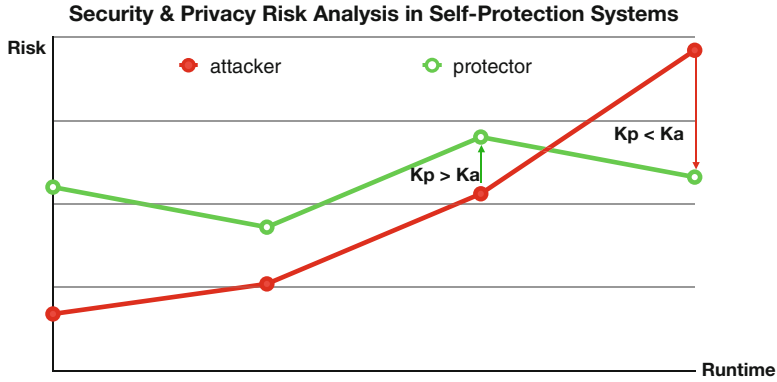

Fig. 6.2 Knowledge adaptation centric to assessing security and privacy risks

typically involve dynamic behaviours; therefore, it is also necessary to consider the behaviour models explicitly.

The uncertainty in self-adaptive systems can be classified into *unknown knowns*, *known unknowns* and *unknown unknowns* [11], where the known unknowns shall be addressed at the runtime, leaving unknown knowns and unknown unknowns to the approaches that perform machine learning, which is beyond the scope of this chapter. Specifically, in this chapter, we would like to address the following research questions:

- Can behavioural models be modified at runtime to reflect the new changes of the known unknowns at design time?
- Can the known unknowns be explicitly defined as parameters on top of the behavioural models at runtime?
- When an attacker can adapt their behaviour according to their knowledge superior to what the protector knows, would self-adaptation capability be misused to hurt security?
- If it cannot be prevented to misuse self-adaptive systems, how can we exert runtime control into the design against this possibility of misuses?

Conceptually, Fig. 6.2 depicts the relationship between the knowledge of defender (K_p) and the attacker (K_a). In principle, when the attacker has more knowledge than the protector, the security and privacy risks of self-adaptive systems are likely to increase. When the defender knows more, the risks can be controlled better. However, the knowledge boundary between the defender and attacker is not always explicitly defined and can change over time. Therefore, the reassessment of the security and privacy risks needs to be performed continuously. Taking the ancient analogy of *spears* (矛) for attacks and *shields* (盾) for protections, the best means for meeting security goals and anti-goals are not staying constant. When the two sides are confronting each other at runtime, the winner is not always predictable unless there is a systematic way to manage the changes.

The key question to ask is, if such analogy holds, whether there is a way to quantitatively access the impact of self-adaptation on security and privacy? When the frontier knowledge is changed, both self-adaptive system and security controls try their best to deal with uncertain behaviours. How to ensure or maintain the level of security when systems are facing such uncertain adaptive behaviours?

In this chapter, we will use example behaviour models to illustrate these challenges and demonstrate the need to expand the knowledge boundary for a self-protection system.

6.1.1 Motivating Examples

To illustrate the problem and the proposed solutions, we show two example systems briefly here to give the context.

6.1.1.1 A PIN Entry Device System

PINs are used for ATM and various smartphones to authenticate users. They are not as hard as online banking protection because the password allowed to use is limited to a few digits. However, such systems are widely used because it demands little memory from users, hence offering a bit more usability. Since it is widely used and simple, we use this example to illustrate the basic concepts in our risks analysis.

6.1.1.2 A Social Media System

Social media such as Facebook are widely used to connect people by posting messages to friends who can pass them onwards to the friends of friends. Privacy, however, it is a key asset to protect so that the information is not passed on the unintended audience. Since the users of social media systems follow their instinct to share posts, it is likely such privacy concerns are violated. The user behaviour-based risk analysis approach proposed in this chapter will be exemplified, again using a simplified behaviour model of a social media system of Facebook.

6.2 Abstract Goal Behaviour Models

In order to perform such an analysis on security and privacy risks, we first introduce the behaviour models for agents, including both human and machine, according to Jackson's abstract goal behaviour models [6].

Definition 6.1 (Behaviour Model) The behavioural model of a domain (or a machine) is represented by a state machine, denoted as a tuple $< S, s_0, T, G, A >$ where S is the set of states, $s_0 \in S$ is an initial state, A is the set of actions on an

alphabet, $T : S \times A \times S$ is a set of A labelled transitions between the states, and $G : T \times \mathcal{B}$ is the set of Boolean guard conditions, indicating whether a transition can be fired.

We assume that the agents have goals that determine their interpretations of the current states of the domains in the world.

Definition 6.2 (Goals) A goal g can be defined as a certain property that holds on the desired states. In other words, given an initial state s_0, the goal of a system can be described by a set of states $s \in S$ such that $g(s)$ is true.

Consider the satisfaction of goals; according to van Lamsweerde [8], there are four typical modes. An *ACHIEVE* goal is described by $\neg g(s_0) \wedge g(s)$, indicating that the goal property was not initially true; a *MAINTAIN* goal is described by $g(s_0) \wedge g(s)$, indicating that initially established property is maintained to be true; a *CEASE* goal is described by $g(s_0) \wedge \neg g(s)$, indicating that the initially established "anti-goal" is no longer true; and an *AVOID* goal is described by $\neg g(s_0) \wedge \neg g(s)$, which avoids the satisfaction of an anti-goal. All these modes can be mapped nicely to security and privacy goals, where the *ACHIEVE* goal of a protector can be regarded as the *CEASE* goal of an attacker and vice versa.

Definition 6.3 (Abstract Goal Behaviours) Since requirements goals are prescriptive on the machine and domains, we can establish the following basic requirements satisfaction argument according to [17]:

$$W, S \models R \tag{6.1}$$

where W and S are nothing but the properties of behaviour models with respect to world context domains and specification of the machine, respectively, while R is the abstract goal behaviours desired by composing these behaviour models.

The intermediate concept of abstract goal behaviours connects the requirements properties with respect to those of the machine domains. When problem domains are biddable or uncertain (e.g., human actors are non-deterministic), we need to handle the uncertainty by considering probabilistic behaviours. In order to quantify these abstract behaviour models, we introduce the notion of risks in terms of likelihood and impact, as follows.

6.3 Risks in Behaviour Models

In this section, we give the definitions of R-DTMC behavioural models and their extension for modelling transparency. Then we provide the technical details of algorithms to compute the security risks by composing the models and the adaptive transparency of risk functions.

Definition 6.4 (Discrete Time Markov Chains (DTMC), Reward DTMC) A *DTMC* extends a state machine by a function $\pi : \delta \to [0, 1]$ that is the probability for a transition in T to be successfully fired. For every state s, the sum of the probability of its outgoing transitions is $\sum_{s'|(s,s')\in\delta} \pi(s, s') = \{0, 1\}$. When the sum is zero, the state is an *absorbing* or *final* state. Furthermore, an R-DTMC extends a DTMC with an impact function $\mathcal{I} : S \to [0, \infty)$ as the reward for reaching a state $s \in S$, typically it indicates the impact of damage on the assets.

Note that in its general form, R-DTMC could associate an impact on transitions as well. In this work, we do not require this level of generality because we have been focusing on the risks of damaging assets at these states, rather than the risks of certain actions on the transitions. By associating the impact with the source state of a transition, our state-only representation of impact is equivalent to associating any impact with the transition.

Definition 6.5 (Traces and Risks) From Definition 6.4, a *trace* $\langle s_0, s\rangle$ from the initial state s_0 to a state s is defined by a sequence of n transitions $(s_k, s_{k+1}) \in \delta$ where $n > 0, k = 0, \ldots, n-1$, and $s_n = s$. From the same pair of states s_0 and s, there could be more than one trace, and these traces may have different lengths. For a given trace $\langle s_0, s_n\rangle$ of length n, the *likelihood* $p(s)$ is defined as follows:

$$p(s) = \prod_{k=0}^{n-1} \pi(s_k, s_{k+1}) \tag{6.2}$$

and the associated *risk* $r^n(s)$ is defined as the product of impact $I(s)$ and likelihood $p(s)$:

$$r^n(s) = I(s) \times p(s), \tag{6.3}$$

which measures how the impact could take effect when the state at the end of the trace is reached from the initial state, at certain likelihood. Considering all possible traces from s_0 to s, the aggregate risk on the state s is given as

$$r^*(s) = \sum_{n=1}^{\infty} \sum_{\langle s_0,s\rangle\in\delta^n} r^n(s). \tag{6.4}$$

One can use a naïve Algorithm 1 to simulate a stochastic decision process which walks on a random transition of each state with respect to the probability distribution of the outgoing transitions. The input also contains two thresholds, t for the total number of traces to simulate and n for the maximal length of the traces before a final state is reached. When there could be infinite length of a trace due to cycles, the simulation forces a trace to terminate when its length is larger than a certain threshold (Lines 2) or when it already has no further transitions (Lines 3–5). Otherwise, the random walk is based on a uniformly distributed random number

Algorithm 1: Compute risks by simulating a stochastic decision-making process through random walks

Data: An R-DTMC (S, δ, s_0, π, $\mathcal{I}$) per Definition 6.4;
t, maximal number of traces to simulate through random walks;
n, maximal length of traces to simulate;
Result: Approximated risks $r^n(s)$ per Definition 6.5 from the simulated traces

1 **for** c:=1 *to* t **do**
2 **for** $l := 1..n$ **do**
3 **if** $\sum_{(s_{l-1},s_k)\in\delta} \pi(s_{l-1}, s_k) = 0$ **then**
4 **break**;
5 **end**
6 $r :=$ random(0, $\sum_{(s_{l-1},s_k)\in\delta} \pi(s_{l-1}, s_k)$) where k indexes outgoing transitions of s_{l-1};
▷ uniformly distributed random number
7 Let $s_l = s_k$ where k is the minimal number so that $r < \sum_{(s_{l-1},s_k)\in\delta} \pi(s_{l-1}, s_k)$;
8 $r^*(s_l) := r^*(s_l) + i(s_l)$;
9 **end**
10 **end**
11 $r^n(s) := r^n(s)/t$ for each state s;
12 return $r^n(s)$;

generator (Line 6). When it falls into the slot by the probabilistic distribution of the outgoing transitions from the previous state s_{l-1}, the corresponding outgoing transition will be assumed (Line 7). On that transition, the risk of reaching the current state s_l will be updated by adding its impact (Line 8).

Finally, the risk is computed as dividing the aggregated impact by the number of simulated traces through random walks, t (Line 11). Note that the chance of selecting an outgoing transition of the previous state is proportional to the probability of the outgoing transitions.

The time complexity of Algorithm 1 in terms of the number of random decisions is $O(tn)$. To get more precision, both t and n need to be larger. Yet when the machine contains cyclic transitions, it is impossible to enumerate all traces.

Depending on the probabilities assigned to the cyclic transitions, the risks in Algorithm 1 are an approximation on the threshold of n, which may not converge to constants when n increases. To illustrate, consider any transitions that form a self-cycle $(s, s) \in \delta$ with $\pi(s, s) = 1$. In such a trace, the state s will be visited n times with the likelihood of 1. Its risk, computed by the simulation, $n \times I(s)$, will increase proportionally to n.

When the cyclic exploration machine gets more complex, however, it is no longer obvious whether the risk computation by simulation converges or not. Even when it converges, a large number of enumerations could be taken to approximate the

risks in order to achieve high precision. The challenge is, given an R-DTMC, there should be a way to tell whether the risks converge or not without lengthy simulations. Furthermore, is there a way to compute the converging risks precisely and efficiently, without all the simulations?

In a matrix form, the likelihood computation can be rewritten as solving the likelihood vector $\mathbf{p}$ to a system of recurrence equations:

$$\begin{aligned} \mathbf{p} &= P\mathbf{p} + \mathbf{c} \\ \mathbf{p} &\geq \mathbf{0} \end{aligned} \tag{6.5}$$

where P is a $n \times n$ transition probabilities square matrix and $\mathbf{c} = (1, 0, \ldots)$ and $\mathbf{p}$ are $1 \times n$ vectors of nonnegative real numbers.

Rewriting this as a linear equation where I stands for the identity matrix of dimension $n \times n$, i.e. $\mathbf{p}I = \mathbf{p}$, we have:

$$(I - P)\mathbf{p} = \mathbf{c} \tag{6.6}$$

The solution of $\mathbf{p}$ can be obtained as:

$$\mathbf{p} = (I - P)^{-1}\mathbf{c} \tag{6.7}$$

When the probability matrix P and the impact vector $\mathbf{i}$ have elements of non-numeric expressions, we call them *symbolic*. When P is not lower-triangular matrix, we call the model *cyclic*, which can still be solved into a risk profile function by applying symbolic LDU decompositions recurrently. Limited by space, we have put the details of algebraic computation of the risk into a technical report,[3] with a proof that when the behavioural model converges (i.e. the necessary and sufficient condition requires a single exit state which does not have any outgoing transitions), the resulting risk profile function can be obtained without simulating on every combination of values in Algorithm 1.

6.4 Running Examples

In [16], we introduced a systematic approach to elicit risk factors from the context diagrams of a software system. The example we used is PED (PIN entry device), where certain security risks have been identified "quantitatively". However, since the quantification was based on natural language processing and CVSS records, it is not yet associated with the behavioural models, hence cannot be applied at runtime.

Here let us first simplify the example so that it is easy to see how risk assessment can be quantified onto the behavioural models.

[3] https://github.com/yijunyu/demo-riskexplore/tree/master/doc

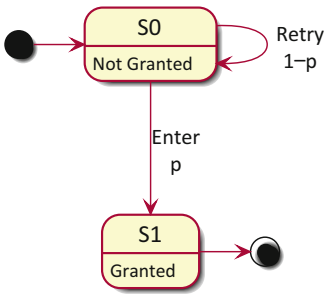

Fig. 6.3 A simple behaviour model for security risks assessment

6.4.1 Security Risks in PIN Access Control

Figure 6.3 illustrates how attackers could gain access to the account after infinite number of trials. Such cyclic behaviour model is quite common in real life; however, existing simulation-based model checkers are not able to detect some flaws in the model.

When the probabilities of the transitions and the impact of the states are unknown, we need to change the way of looking at them as numeric values, but as algebraic symbols (i.e. known unknowns) instead.

Assume that the overhead for login was $-O$, which rewards the attacker by the value of bank account V, then the risk of loss is estimated to be

$$-O/p + V \tag{6.8}$$

When p is small enough, the following condition could provide some relative assurance of the system security:

$$O/p > V \tag{6.9}$$

This explains why an effective policy for preventing denial-of-service attacks is to introduce some overhead to the users while logging in to the system, so that it is not worthwhile to try indefinitely.

6.4.2 Privacy Risks in Social Networks

While social networks systems are used by individual users, they may choose to share posts to the friends, with a non-negligible probability that the friends may share the posts further to unwanted or undesirable audience. The trade-offs between sharing and not sharing, with respect to social benefits such as likes and resharing, are frequent decisions to be made by the individual. The rationale of such decisions are typically risk assessments on the basis of simulating the effect of leaking the private information to unintended audience [14].

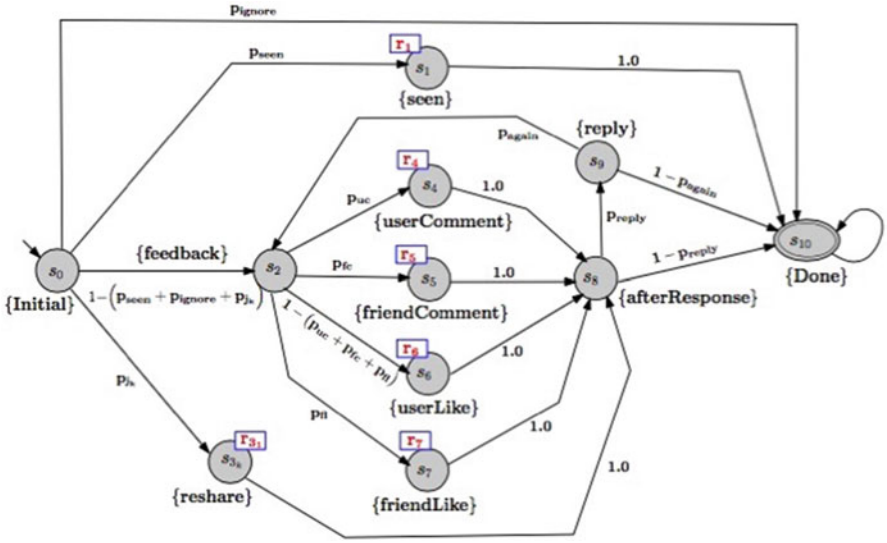

Fig. 6.4 A privacy risk assessment model taken from [10]

Recently, the original work in [14] has been extended to introduce a much more complex behavioural model of sharing [10] by introducing inductive machine learning techniques similar to those of recommendation systems. In other words, patterns of groups emerging from the social circles are learnt by different individuals, while they are making similar decisions.

However, simulation-based approaches are inherently incomplete. For example, the risk assessment model in Fig. 6.4 is a little bit more complicated than the security one we discussed earlier. It is based on the individual's decisions to share post on a social network and the estimation of the risk exposure to the audience if the information is sensitive.

The behaviour model was built using PRISM [7], but the algebraic symbols are computed differently here. After applying our risk explorer tool,[4] the risk profile function can be obtained as such:

```
pseen*r1+pjk*r3-(r4*puc*(pjk-(1-pseen-pignore)
-pagain*preply*pjk))/(1-pagain*preply*pfl
+pagain*preply*(pfl-(1-puc-pfc))-pagain*preply*pfc
-pagain*preply*puc)-(r5*pfc*(pjk-(1-pseen-pignore)
-pagain*preply*pjk))/(1-pagain*preply*pfl
+pagain*preply*(pfl-(1-puc-pfc))-pagain*preply*pfc
-pagain*preply*puc)+(r6*(pfl-(1-puc-pfc))*(pjk-
(1-pseen-pignore)-pagain*preply*pjk))/(1
-pagain*preply*pfl+pagain*preply*(pfl-(1-puc-pfc))
-pagain*preply*pfc-pagain*preply*puc)-(r7*pfl*(pjk
```

[4] https://github.com/yijunyu/demo-riskexplore

```
-(1-pseen-pignore)-pagain*preply*pjk))/(1
-pagain*preply*pfl+pagain*preply*(pfl-(1-puc-pfc))
-pagain*preply*pfc-pagain*preply*puc)
```

where the following determinant condition has to be satisfied; otherwise the behavioural model will not converge to a solution:

```
0<1-pagain*preply*pfl+pagain*preply*(pfl-(1-puc-pfc))

  -pagain*preply*pfc-pagain*preply*puc
```

By applying an optimisation algorithm to minimise the risk profile function, e.g. differential evolution optimisation [9], it is possible to obtain a near-optimal solution in less than 1 min.

For example, when $r1 = r2 = r3 = r4 = r5 = r6 = r7 = 1$, the lowest risk of 0.012 can be achievable when $pseen = 0.006239582$, $pignore = 0.987266657$, $pjk = 0.003001324$, $puc = 0.115949677$, $pfc = 0.446085095$, $pfl = 0.131866686$, $preply = 0.003728548$ and $pagain = 0.048901284$.

6.5 Discussions

Of course, this combination of the known unknowns for "minimum" risks is derived without considering any constraints. With more knowledge at runtime, either for the protector or for the attacker, the minimal risk would not look the same because they would see these unknowns differently.

In principle, both normal user and attackers can be modelled as biddable domains, in which not all decisions are deterministic and not all states are explicit. In other words, unless we are the attackers, such models are just intellectual guesses. Nonetheless, having a model allows us to estimate the risks of actions of individual agents.

We assume that the attackers and the defenders have different knowledge of the system. In other words, through observations, the attacker could realise some vulnerabilities before the defenders knows, and the defenders certainly could know some internal designs that the attacker may not know.

In the following, we show an example where the attacker knows a vulnerability before it manifests to the public.

Suppose the protector initially assume that the PIN code protection used in the system is uniformly distributed and to guess correctly the 4 digits one would have to try 10,000 combinations in the worst case and 5000 in the average case. However, through key loggers or other means, attackers could estimate the distribution of probability so that p increases to 0.5. In that case, the current behavioural model could no longer offer sufficient protection since the risk increases dramatically. Similarly, if the protector knows that the account has $0 in value, while the attacker does not, it becomes easier for the protector to set up a trapping "honey pot" in order to catch such reckless attackers. In this case attackers would face higher risks.

6.6 Summary

In this chapter, we have articulated the need to quantify the risks for self-protection, i.e. offering both protectors and attackers' perspectives in assessing the risks. The known unknowns, in this work, manifest as symbolic probabilistic variables appearing on the guard condition of transitions in behavioural models. We have also used two examples from security and privacy application domains to illustrate the advantage of such quantified risk exploration.

Note that the work of risk exploration is an ongoing research effort, where we have developed open-source tools for colleagues to use and compare with our results https://github.com/yijunyu/demo-riskexplore. A guide tour of risk exploration can be found in the tutorial [15].

In the future, we hope to improve the efficiency of our quantitative risk exploration tool so that self-protection systems could be armed with the runtime behaviour models to define the set points for efficient self-adaptation.

References

1. Barth, A., Datta, A., Mitchell, J.C., Nissenbaum, H.: Privacy and contextual integrity: framework and applications. In: Proceedings of the 2006 IEEE Symposium on Security and Privacy, SP'06, pp. 184–198. IEEE Computer Society, Washington, DC (2006). https://doi.org/10.1109/SP.2006.32
2. Brun, Y., Di Marzo Serugendo, G., Gacek, C., Giese, H., Kienle, H., Litoiu, M., Müller, H., Pezzè, M., Shaw, M.: Engineering Self-Adaptive Systems through Feedback Loops, pp. 48–70. Springer, Berlin/Heidelberg (2009). https://doi.org/10.1007/978-3-642-02161-9_3
3. Chen, B., Peng, X., Yu, Y., Zhao, W.: Requirements-driven self-optimization of composite services using feedback control. IEEE Trans. Serv. Comput. **8**(1), 107–120 (2015). https://doi.org/10.1109/TSC.2014.2298866
4. Iglesia, D.G.D.L., Weyns, D.: Mape-k formal templates to rigorously design behaviors for self-adaptive systems. ACM Trans. Auton. Adapt. Syst. **10**(3), 15:1–15:31 (2015). https://doi.org/10.1145/2724719
5. ISO/IEC: Iso/iec 25010 system and software quality models. Technical report (2010)
6. Jackson, M.: System behaviours and problem frames: concepts, concerns and the role of formalisms in the development of cyber-physical systems. In: Dependable Software Systems Engineering, pp. 79–104 (2015). https://doi.org/10.3233/978-1-61499-495-4-79
7. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: verification of probabilistic real-time systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) Proceedings of 23rd International Conference on Computer Aided Verification (CAV'11), Snowbird. LNCS, vol. 6806, pp. 585–591. Springer (2011)
8. van Lamsweerde, A.: Goal-oriented requirements engineering: a guided tour. In: 5th IEEE International Symposium on Requirements Engineering (RE 2001), 27–31 Aug 2001, Toronto, p. 249 (2001). https://doi.org/10.1109/ISRE.2001.948567
9. Mullen, K., Ardia, D., Gil, D., Windover, D., Cline, J.: DEoptim: an R package for global optimization by differential evolution. J. Stat. Softw. **40**(6), 1–26 (2011). http://www.jstatsoft.org/v40/i06/
10. Rafiq, Y., Dickens, L., Russo, A., Bandara, A.K., Yang, M., Stuart, A., Levine, M., Calikli, G., Price, B.A., Nuseibeh, B.: Learning to share: engineering adaptive decision-support for

online social networks. In: Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE '17), IEEE Press, Piscataway, pp. 280–285 (2017)
11. Sutcliffe, A., Sawyer, P.: Requirements elicitation: towards the unknown unknowns. In: 2013 21st IEEE International Requirements Engineering Conference (RE), Rio de Janeiro, pp. 92–104 (2013). https://doi.org/10.1109/RE.2013.6636709
12. Tun, T.T., Bandara, A.K., Price, B.A., Yu, Y., Haley, C., Omoronyia, I., Nuseibeh, B.: Privacy arguments: analysing selective disclosure requirements for mobile applications. In: 2012 20th IEEE International Requirements Engineering Conference (RE), Chicago, pp. 131–140 (2012)
13. Warren, S.D., Brandeis, L.D.: The right to privacy. Harvard Law Rev. **4**(5), 193–220 (1890). http://www.jstor.org/stable/1321160
14. Yang, M., Yu, Y., Bandara, A.K., Nuseibeh, B.: Adaptive sharing for online social networks: a trade-off between privacy risk and social benefit. In: 13th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, TrustCom 2014, Beijing, 24–26 Sept 2014, pp. 45–52 (2014). https://doi.org/10.1109/TrustCom.2014.10
15. Yu, Y.: Risk assessment using early requirements models: a guided tour. In: 25th International Requirements Engineering Conference, Tutorial, Lisbon (2017)
16. Yu, Y., Franqueira, V.N.L., Tun, T.T., Wieringa, R., Nuseibeh, B.: Automated analysis of security requirements through risk-based argumentation. J. Syst. Softw. **106**, 102–116 (2015). https://doi.org/10.1016/j.jss.2015.04.065
17. Zave, P., Jackson, M.: Four dark corners of requirements engineering. ACM Trans. Softw. Eng. Methodol. **6**(1), 1–30 (1997). https://doi.org/10.1145/237432.237434

Chapter 7
Experimenting with Adaptation in Smart Cyber-Physical Systems: A Model Problem and Testbed

Vladimir Matena, Tomas Bures, Ilias Gerostathopoulos, and Petr Hnetynka

Abstract The chapter focuses on experimentation with adaptation in the field of smart cyber-physical systems (sCPS). In particular, it provides a model problem that features a coordination of autonomous cleaning robots. The model problem is accompanied with a testbed which allows the execution of the model problem along with custom adaptation logic. The testbed can be executed as a simulation of multiple robots running or deployed on an actual TurtleBot robot. Both the simulated and actual deployment environment are based on the same software stack. The offered simulation is precise timing-, bandwidth-, and mobility-aware and brings together a ROS-based Stage simulation of a swarm of robots and OMNeT++-based simulation of 802.15.4 wireless network, while the actual deployment is based on the TurtleBot robotic platform. The adaptation business logic is based on the DEECo component model and points to specific places, where the user code can be easily plugged in.

7.1 Introduction

Smart cyber-physical systems (sCPS) are distributed and decentralized systems that closely cooperate with their physical environment by sensing and actuating [9]. A characteristic feature of sCPS is that they exhibit a high level of "intelligence" in terms of opportunistic cooperation, dynamic self-organization, self-healing, and self-adaptation [6]. As such, sCPS are regarded as vital for building applications for smart mobility, smart energy grids, ambient assisted living, smart cities, etc.

V. Matena (✉) · T. Bures · P. Hnetynka
Faculty of Mathematics and Physics, Charles University in Prague, Prague, Czech Republic
e-mail: matena@d3s.mff.cuni.cz; bures@d3s.mff.cuni.cz; hnetynka@d3s.mff.cuni.cz

I. Gerostathopoulos
Faculty of Mathematics and Physics, Charles University in Prague, Prague, Czech Republic

Fakultät für Informatik, Technische Universität München, Munich, Germany
e-mail: iliasg@d3s.mff.cuni.cz

Y. Yu et al. (eds.), *Engineering Adaptive Software Systems*,
https://doi.org/10.1007/978-981-13-2185-6_7

Software engineering of sCPS is largely an open challenge, as sCPS combine autonomous decentralized cooperative behavior, with concerns of real-time, limited communication, dependability, etc. The lack of software engineering support also applies to self-adaptation [7], which is a central feature of sCPS, crucial for coping with the uncertain environments in which sCPS operate.

While there is a large body of knowledge for experimenting with adaptation in the context of enterprise services and other traditional software systems, there is rather a vacuum in terms of knowledge and especially tools for experimenting with adaptation in sCPS. This is in our view because sCPS combine multiple relatively distinct disciplines (real-time, control, networking, agents, learning, data-analysis, etc.) [4]. This consequently requires engineering approaches and tools for sCPS to build synergies between the disciplines and support the mutual interplay of the concerns.

In this chapter,[1] we partially address the problem of development of self-adaptive sCPS by providing a model problem and testbed for experimenting with, comparing, and developing new adaptation solutions pertinent to sCPS.

In particular, the model problem and testbed provide challenges in coordination of autonomous robots with the interplay of concerns of (a) realistic communication (i.e., communication limited by bandwidth and subject to latencies), (b) real-time control, and (c) decentralized operation.

To enable fast prototyping, the testbed abstracts robots as autonomous components (implemented in Java) and allows describing robot communication via dynamic collaboration groups. It also points to specific places in the code where adaptation logic can be plugged in and provides metrics for evaluating the plugged-in adaptation. Thus, together, the model problem and the testbed provide a concrete ready-to-use benchmark for experiments in the relatively new field of sCPS.

Either the implementation can be executed as a simulation, or it can be directly deployed to actual robots (currently, the implementation out of the box supports the TurtleBot robots[2]).

The chapter is organized as follows. Section 7.2 describes the model problem in detail. Section 7.3 presents the testbed from both the user perspective and also implementation point of view. Section 7.4 describes a sample adaptation we have used for evaluating the testbed and further discusses lessons learned and limitations. Section 7.5 briefly details the structure of the provided testbed (detailed instructions are packaged together with the testbed). Section 7.6 discusses related work, while Sect. 7.7 concludes the chapter by summarizing the contributions.

[1] It is based on material included in a SEAMS 2016 publication by the same authors [5].

[2] http://www.turtlebot.com/

7.2 Model Problem

The model problem provided by our testbed is the "Autonomous Cleaning Robots Coordination" (ACRC) problem. In ACRC, a number of cleaning robots is deployed in in-door space consisting of corridors and multiple office rooms (see Fig. 7.1).

Every robot is equipped with a camera which provides depth information. The robots use the cameras to observe obstacles (other robots, walls, etc.) and for navigation, by means of Adaptive Monte Carlo Localization (AMCL). Robots are equipped with a map of the place that they are supposed to clean. This map is used in the AMCL-based navigation, which works by comparing a depth scan with the map.

Robots are capable of limited communication using an IEEE 802.15.4 transceiver (with approx. 10 m direct visibility range), which allows building mobile ad hoc networks. This means that robots can exchange data only when they are close to one another. Robots can extend the communication range by acting as proxies that rebroadcast messages further. Generally, however, no global communication can be assumed as situations when no proxy is close enough or too much interference exists are rather often.

The basic software of the robots is formed by the Robot Operating System[3] (ROS), which is the de facto standard set of libraries and services for building open-source robotic platforms.

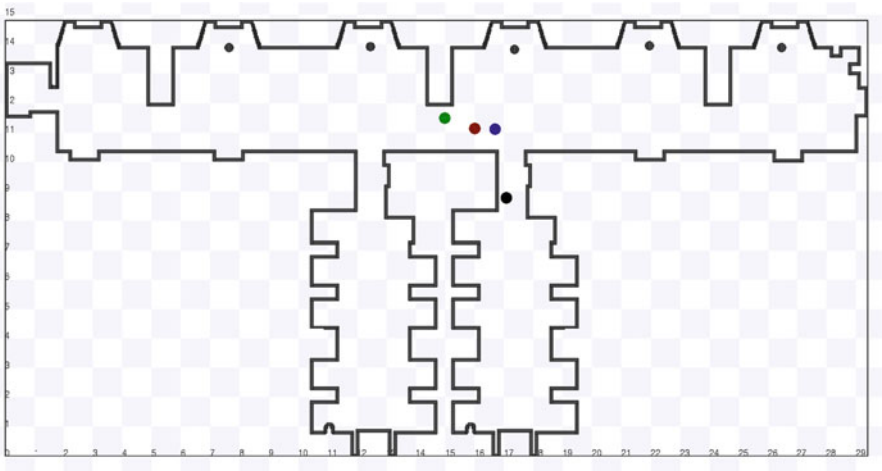

Fig. 7.1 A visualization of the model problem

[3] http://wiki.ros.org/

7.2.1 Operation and Adaptation Challenges

Each robot is initially given its own set of places it is supposed to visit and clean. In the naïve solution, which can be considered as the baseline, robots act completely independently of one another (i.e., they do not communicate nor coordinate) and visit places on their list in the given order.

Due to the complexity of the environment and the deficiencies in the ROS stack (which we consider as a black-box component that is given and one has to live with), the naïve solution gives rise to multiple problems:

- A robot has only an approximation of its position and orientation. Often, especially when other robots are present nearby, the AMCL localization fails as the depth scans (which include other robots) cannot be matched with the known map. As a result, the robot navigation becomes very imprecise and sometimes, when in dense traffic, fails completely, and the robot stops.
- The navigation module in a robot sometimes fails to find a route to the destination because other robots moving by obstruct it. As the result the robot stops.
- Due to physical space constraints, robots often get to a deadlock situation – e.g., when one robot wants to enter the office and another wants to exit it. The result is again that the robots stop to avoid collision. (Note that this is a different situation to the previous point, where the failure to find a way is only transient. In this case, however, it persists until the deadlock is explicitly solved.)

Generally, each of these problems can be solved by pointing the robot to the right direction. However, it practically turns out to be quite difficult to (1) distinguish the cause of the problem and (2) to know where to navigate the robot to recover it from the failed state.

Though these problems could be targeted by modifying ROS, our experience with extending and customizing ROS shows that a more practically viable solution is to regard ROS as a black-box and build an adaptation layer over it. As such, the robotic scenario constitutes an excellent case for adaptation. (Of course, this is by no way a criticism of ROS, which itself is the most comprehensive open-source solution for robotics. It is more an acknowledgment of the complexity inherently connected with developing systems that perform in and interact with real environments.)

To remediate the deficiencies of the baseline solution, the adaptation layer has generally free access to the robot navigation. In particular, it can obtain estimates of the position and can sense whether the robot moves. Based on this, it can:

- manipulate the queue of locations to be visited (destinations),
- pause the robot and command the robot to move to any place on the map.

Additionally, the adaptation layer on one robot may communicate with the adaptation layers of other robots to realize more complex adaptation strategies via cooperation.

The adaptation however comes with another set of problems once we try not only to recover the robots from failures and deadlocks but also optimize the overall

performance of the system. Clearly, by reordering the locations to be visited and by transferring the responsibility of cleaning a place from one robot to another, the system can highly optimize itself. Theoretically, it can even get to a point when no collisions happen because robots exchange their destinations in such a way that they do not interfere. This is however subject to multiple problems, which can be regarded as additional adaptation challenges:

- The uncertainty in location makes planning not completely reliable.
- Communication range is limited, which means that robots in different rooms cannot communicate directly but only through proxies (if present), which have to be located in the corridor close to the office entrances.
- The communication is subject to latencies and unreliability (due to interference) which makes it impossible for a robot to have an up-to-date knowledge of the global state of the system and disallows strong synchronization among robots.

7.2.2 Solution Comparison Dimensions

Having the adaptation logic in place, various metrics can be considered for evaluation and comparison of different adaptation strategies (solutions to ACRC). We list below metrics which we found useful in our experiments with ROS-controlled robots. Note that since ACRC contains random elements and non-determinism, the evaluation of a solution requires multiple simulation runs of ACRC and statistical evaluation (e.g., by statistical testing of sample means or quantiles).

Time to complete all the tasks (i.e., visit and clean all locations assigned to the robots at the start) can be regarded as the basic metric when we assume that the evaluated solution is able to make the robots complete all their tasks. Our experience showed that this is more difficult than it appears to be. For evaluating partial successes, we thus suggest the following metrics.

Number of cleaning tasks that were completed This covers situations when time limit for completion expires or when the system itself realizes that certain locations cannot be cleaned – e.g., if a robot gets stuck in a room entrance and any attempts to move the robot out of the way fail.

Total running time till system completes or gives up This can be used as a metric complementary to the above one, to reward solutions which possess the ability to recognize that certain problems cannot be solved. It can serve to resolve ties in case two solutions are statistically similar (e.g., a statistical test cannot reject the hypothesis of the two solutions that have the same average number of cleaning tasks completed).

7.3 Testbed

The provided testbed allows for easy experimentation with adaptation techniques and algorithms for the ACRC problem. The model problem is implemented on top of ROS, and both the adaptation logic and the adapted system are specified using DEECo, which is a component model for sCPS. Details on the technical architecture are given in Sect. 7.3.5.

The testbed offers two modes of deployment and execution. The first mode is a simulation of a swarm of TurtleBots solving the ACRC problem. This mode is primarily suitable for early stages of development and/or for conducting quantitative measurements. The second mode allows for actual deployment using real TurtleBots.

The *simulation mode* is implemented on top of the Stage simulator, which is tightly integrated with ROS. The Stage simulator is capable of simulating robot physics, movement, laser scan sensing, and odometry readings. Currently the included Stage is configured to simulate TurtleBots only, but it is possible to change robot shape, movement model, and sensors to match different robots as well.

The Stage simulator is further extended by a custom integration of the OMNeT++ network simulation into ROS – called ROSOMNeT++[4] – which enables sending and receiving IEEE 802.15.4 packets using ROS facilities.

For the *actual deployment,* it is necessary to equip each TurtleBot with an onboard computer and wireless network interface. Regarding the onboard computer, any average contemporary machine is suitable (we tested it with the Intel P9600 CPU and 6 GB of RAM, and such a configuration was completely sufficient). For the wireless network interface, an external microcontroller with an IEEE 802.15.4 module is expected. The testbed has been tested with and is prepared for the STM32F4[5] board equipped with the extension board[6] and the BEE click[7] module. All of them are off-the-shelf components.

ROS already contains modules, which serve as drivers for TurtleBot, and we have developed extensions to support the IEEE 802.15.4 network. In particular, we have developed two projects. The beeclickarm[8] provides the firmware and Java interface for the used microcontroller, while the beeclickarmROS[9] exports features of the beeclickarm as ROS topics and services.

The detailed instructions about hardware installation and deployment are available in the testbed's README file.[10]

[4] https://github.com/d3scomp/ROSOMNeT

[5] http://www.st.com/stm32f4

[6] http://www.mikroe.com/stm32/stm32f4-discovery-shield

[7] http://www.mikroe.com/click/bee

[8] https://github.com/d3scomp/beeclickarm/tree/robot-additions

[9] https://github.com/d3scomp/beeclickarmROS

[10] https://github.com/d3scomp/deeco-adaptation-testbed

The testbed seamlessly supports both deployment modes; the simulated and actual devices are accessible via the same ROS interface, and thus no changes at user code are required when switching between the deployment modes.

7.3.1 Modeling Concepts for Decentralized Coordination

The robots' behavior is developed using DEECo [2], which is a component model and framework for developing complex sCPS. DEECo is based on concepts of *ensemble-based component systems* (EBCS) (designed primarily in the scope of the EU FP7 ASCENS project[11]). In EBCS, a system is modeled as a set of dynamic cooperation groups of software components – *ensembles*. DEECo itself is an abstract component model; however it comes with two implementations – one in Java[12] (JDEECo) and one in C++[13] (CDEECo). In the testbed, we use JDEECo as we found Java easier for prototyping the components.

A component in DEECo is represented by its data (*knowledge* in EBCS) and its tasks (*processes* in EBCS). Figure 7.2 shows a code skeleton of the baseline implementation of the robot component in JDEECo. JDEECo-specific constructs are expressed using an internal domain-specific language (DSL) defined via Java annotations. A component is defined as a plain Java class annotated with the @Component annotation. Component's knowledge is defined as Java class fields (lines 3–10 in Fig. 7.2). Knowledge that is not supposed to be shared with other components via ensembles (as described below) is marked as @Local. Component's fields are manipulated by the component's processes (e.g., lines 13–33). Processes are defined as, respectively, annotated static Java class methods. Processes are either periodically executed or event-triggered (i.e., commonly as a reaction to knowledge change). This is determined by the annotation attached to the process.

Typically, processes involve sensing, computation, mutation of the component knowledge fields, and actuating. The signature of the process defines which knowledge fields are read/written (as in/out/in-out). Technically, the processes are scheduled by JDEECo runtime, which also takes care of thread-safe retrieval of component's knowledge to be used by a process and storing of the process results back in component's knowledge.

Figure 7.2 lists the processes defined in the baseline implementation of the cleaner robot as provided by ACRC. These are (i) setting the next destination, (ii) reading the position, (iii) reporting the status, and (iv) controlling the movement of the robot.

Communication between components is in DEECo modeled by *ensembles*. An ensemble dynamically determines which components are in the communication

[11] http://ascens-ist.eu/

[12] http://github.com/d3scomp/JDEECo

[13] http://github.com/d3scomp/CDEECo

```
1. @Component
2. public class CleanerRobot {
3.   public String id;
4.   public Position destination;
5.   public Position position;
6.   public State state;
7.   @Local public Long blockedCounter;
8.   @Local public Long noPosChangeCounter;
9.   @Local public Position oldPosition;
10.   @Local public List<Position> route;
11.   ...
12.
13.   @Process
14.   @PeriodicScheduling(period = 500)
15.   public static void setDestination(
16.      @InOut("destination")
       ParamHolder<Position> destination,...)
17.   {...}
18.
19.   @Process
20.   @PeriodicScheduling(period = 100)
21.   public static void  sense( @Out("position")
    ParamHolder<Position> position,
    @In("positioning") Positioning positioning)
22.   {...}
23.
24.   @Process
25.   @PeriodicScheduling(period = 1000)
26.   public static void reportStatus( @In("id")
    String id, ...)
27.   {...}
28.
29.   @Process
30.   @PeriodicScheduling(period = 2000)
31.   public static void driveRobot(
    @In("position") Position pos,
    @In("positioning") Positioning positioning,
    @In("destination") Position destination,
    @InOut("curDestination")
32.      ParamHolder<Position> curDestination,...)
33.   {...}
34. }
```

Fig. 7.2 Model of ACRC baseline in JDEECo

group via a membership condition. Topologically, an ensemble in JDEECo is a star featuring one *coordinator* and multiple *members*. The communication within ensembles is implicit, i.e., the ensemble defines an exchange method, which performs knowledge exchange among components grouped in the ensemble (i.e., copying data from a knowledge field of one component to a knowledge field of another component).

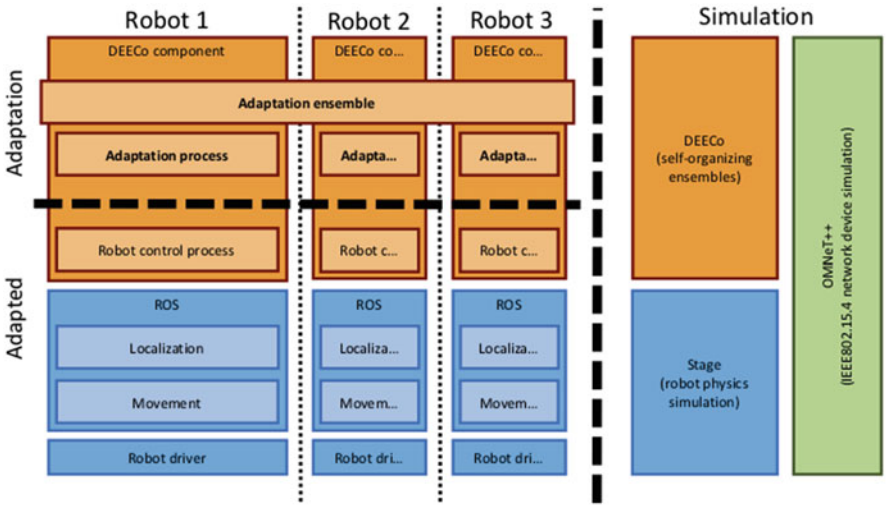

Fig. 7.3 Adaptation architecture

The baseline implementation does not involve any ensembles. However, ensembles are to be exploited for decentralized coordination of adaptation across several robots. This is demonstrated in Fig. 7.12, where an ensemble for location exchange is given. It is established between robots which are close to each other, and both of them are stuck. The ensembles are defined again as plain Java classes with annotations. The membership and exchange methods are periodically executed (with prescribed period – line 2), and their parameters specify the read/written knowledge fields of particular components (prefixes *coord-* and *mbr-* are used to identify coordinator and member role, respectively).

The architectural view of the adaptation is depicted in Fig. 7.3, which shows the split into "adapted" and "adaptation" layers. The adapted layer consists of robot drivers, ROS modules, and business logic implemented as DEECo processes. The adaptation layer is implemented by DEECo constructs. In case of local adaptations, which do not take multiple robots into account, the adaptation is implemented as a DEECo process. In more advanced cases when the adaptation layers of multiple robots need to cooperate, a DEECo ensemble is used to implement the adaptation logic.

7.3.2 Setup

The testbed models the ACRC problem via DEECo component model (in detail in Sect. 7.3.1). In particular, it represents each robot as an instance of the *CleanerRobot* component and provides its baseline behavior (i.e., the base-level subsystem [13])

```
1. ROSSimulation ros = new ROSSimulation(SIM_SRV_ADDRESS,
2.     11311, SIM_SRV_ADDRESS, "corridor", 0.02, 100);
3. DEECoSimulation realm = new DEECoSimulation(ros.getTimer());

4. realm.addPlugin(Network.class);
5. realm.addPlugin(DefaultKnowledgePublisher.class);
6. realm.addPlugin(KnowledgeInsertingStrategy.class);
7. realm.addPlugin(BeeClick.class);

8. for (int i = 0; i < NUM_ROBOTS; ++i) {
9.     final String name = "Collector" + i;
10. List<Position> garbage = generateGarbage(NUM_GARBAGE);
11.     Positioning positioning = new Positioning();
12.     DEECoNode robot = realm.createNode(i, positioning,
13.         ros.createROSServices(colors[i]), positionPlug[i]);
14.     robot.deployComponent(new CleanerRobot(name, positioning,
15.         ros.getTimer(), garbage,...));
16.     robot.deployEnsemble(DestAdoptionEnsemble.class);
17.     robot.deployEnsemble(AdoptedDestRemoveEnsemble.class);
18. }

19. realm.start(600_000);
```

Fig. 7.4 Deploying multiple robots in simulation

in Java. The testbed provides a well-defined place in the component where the adaptation logic is to be plugged in (i.e., the reflective subsystem). Technically, this is done by introducing additional periodic processes to the Collector Robot component and additional ensemble specifications (e.g., see Sect. 7.4.1). There is no difference between the setup for the simulator deployment and the actual TurtleBot deployment; only the initialization and launching differ as follows.

Simulation setup In Fig. 7.4 the code responsible for initializing the simulation is shown. Lines 1–3 establish DEECo simulation using ROS, lines 4–7 load DEECo plug-ins shared by all robots, the loop on lines 8–18 deploys robots, and finally line 19 runs simulation for 600 s. The simulation is configured by the number of robots, their initial positions, and the map of the environment. The testbed comes with one map that comprises a corridor and two offices. Custom maps can be provided as PNG files similar to the one shown in Fig. 7.1.

Actual TurtleBot setup Instead of deploying all the robots at once (as in the case of simulation), the deployment code depicted in Fig. 7.5 deploys a single cleaning robot component on a single actual robot. Thus, it is necessary to run the code on each individual robot. Lines 1–4 are responsible for establishing the DEECo system using wall timer and actual robot ROS interface, lines 6–11 deploy a DEECo node with all required plug-ins, lines 13–20 are responsible for deployment of the cleaning robot component and ensembles, and finally line 22 runs the system and blocks forever as the real deployment has no time limit.

```
1. WallTimeTimer wallTimer = new WallTimeTimer();
2. RosServices rosServices = new RosServices(
3.     System.getenv("ROS_MASTER_URI"),
4.     InetAddress.getLocalHost().getHostName());
5.
6. DEECoNode node = new DEECoNode(ROBOT_ID, wallTimer,
7.     new Network(),
8.     new BeeClick(),
9.     new DefaultKnowledgePublisher(),
10.     new KnowledgeInsertingStrategy(),
11.     rosServices, positionPlugin);
12.
13. final String name = "Collector" + ROBOT_ID;
14. garbage = generateGarbage();
15. Positioning positioning = new Positioning();
16. node.deployComponent(new CleanerRobot(name, positioning,
17.     wallTimer, garbage,...));
18.
19. node.deployEnsemble(DestinationAdoptionEnsemble.class);
20. node.deployEnsemble(AdoptedDestinationRemoveEnsemble.class);
21.
22. wallTimer.start();
```

Fig. 7.5 Single actual robot deployment

Further guidelines on deployment as well as the whole source code of the testbed can be found on GitHub.[14]

7.3.3 Debugging

The testbed enables usage of several debugging tools that can be used to observe the system in order to deploy adaptation techniques as well as debug existing adaptation code. The testbed is using ROS topics to control the simulated robot using standard messages described by ROS. Thus it is possible to use ROS tools to inspect and visualize messages in the system at no extra cost. These can be used to obtain a robot-centric view of the system (as displayed in Fig. 7.6) and thus realize what is wrong at a local level. In the following paragraphs, the most important tools are briefly described.

Stage visualization The primary output of the testbed is the direct visualization of the scenario shown in Fig. 7.1 (the visualizer itself comes with the Stage robot simulator – Sect. 7.3.5). Via it, the user can observe the movement of the robots in real time. Black lines represent walls and other obstacles impenetrable for the

[14] https://github.com/d3scomp/deeco-adaptation-testbed

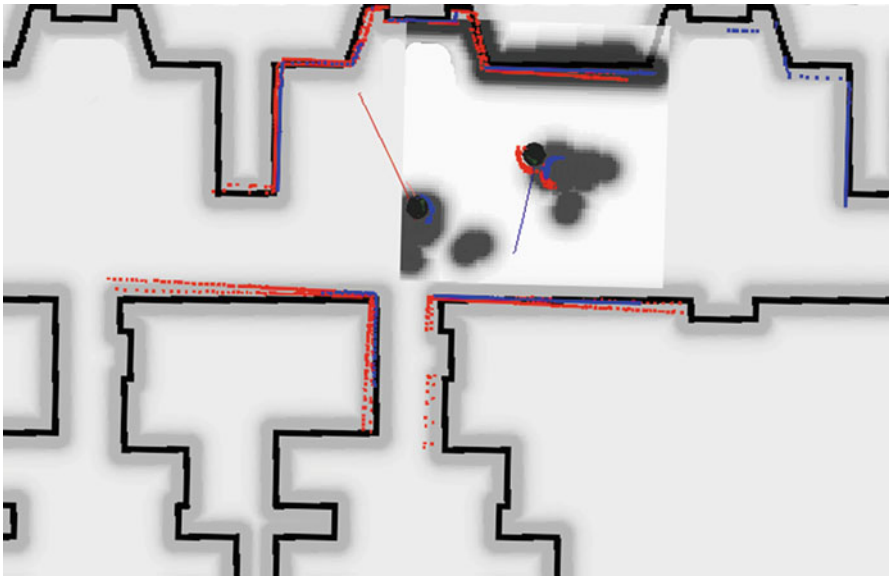

Fig. 7.6 Robots' perception of the environment

robots (i.e., the map provided to the testbed). The colored dots represent the robots as located in the simulated system. Thus the output of the Stage visualization is global view of the systems' ground truth data.

Logging ROS messages As mentioned above robot control is ROS based on sending messages. Fortunately those can be printed to command-line or a file for later processing by plotting or using statistical tools. ROS defines how different datatypes are represented as text, thus printing robot location requires no extra output formatting as shown in Fig. 7.7.

Robot Visualizer (RViz)[15] This tool provides a convenient way to observe a robot's view of the environment by displaying ROS messages in a 2D or 3D. Most of the messages used in the system are directly understood by RViz, so that having data visualized is as easy as choosing the correct data source.

The real power of RViz is the visualization of the data from real robot. Figure 7.8 shows RViz visualization of data from the TurtleBot deployed in a real environment. The background is a static map of the environment which is used for long-range planning. The colored rectangle around the robot is a local map capturing temporary obstacles detected by distance scanner such as chairs and persons. Below the 3D model of the robot, a cloud of green arrows used by AMCL to guess robot location

[15] http://wiki.ros.org/rviz

Fig. 7.7 Printing ROS location message

```
1. $ rostopic echo /robot_0/amcl_pose
2. header:
3.   seq: 78
4.   stamp:
5.     secs: 91
6.     nsecs: 900000000
7.   frame_id: /map
8. pose:
9.   pose:
10.    position:
11.      x: 13.901734935
12.      y: 11.8805620695
13.      z: 0.0
14.    orientation:
15.      x: 0.0
16.      y: 0.0
17.      z: 0.955308313164
18.      w: 0.29561127651
```

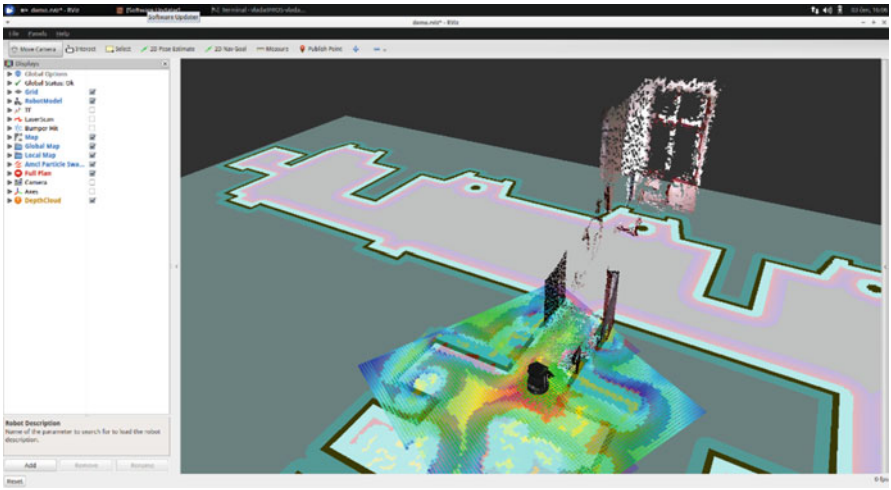

Fig. 7.8 RViz using data from real robot

is visible. Finally a depth image captured by onboard camera is displayed in 3D in order to help guess how guessed location matches reality.

rqt_plot[16] Working on top of ROS messages, an rqt_plot tool can generate plots of various messages in real-time and store them for later use. The output of this tool is depicted in Fig. 7.9.

[16] http://wiki.ros.org/rqt_plot

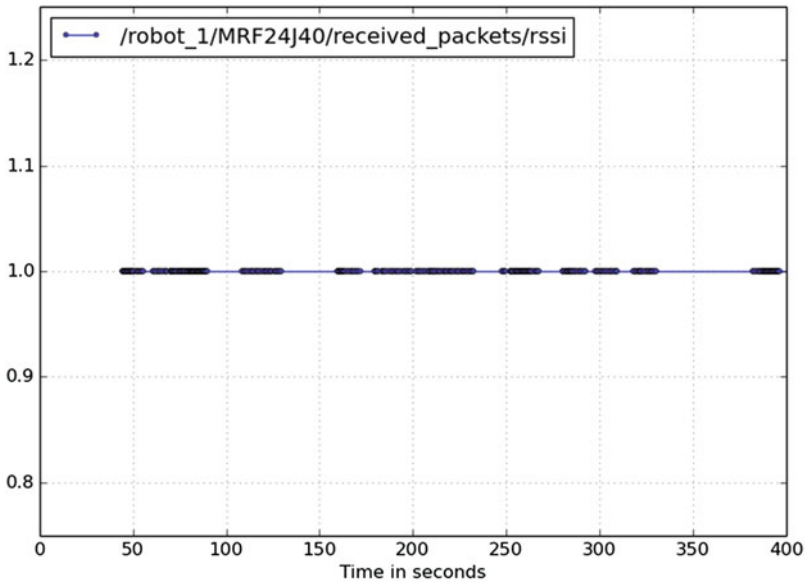

Fig. 7.9 rqt_plot showing MANET packet arrivals

ROS Bags[17] ROS has an ability to record all messages in the system into a file, which can be used for offline analysis. All the aforementioned tools using ROS messages can work on top of replayed messages recorded during simulation or actual system execution. For instance, it is possible to visualize trajectories of the robots and replay the visualization over and over.

This feature is important for recording simulation runs as it saves time needed to execute the same simulation repeatedly. It is even more important for the actual deployments as it is in fact impossible to execute a scenario repeatedly with the exactly same results.

Eclipse debugger[18] As the testbed and all the adaptation code are written in Java, it is possible to run the testbed as a Java application directly from Eclipse IDE and thus use all debugging features of Eclipse. This is possible for both the simulation and actual run. The limitation here (stemming from the soft real-time nature of ROS) is that ROS continues running even if jDEECo and the adaptation logic are paused by debugging. However, this is typically not a problem due to the fact that jDEECo controls ROS essentially only by setting robot waypoints. As such, if jDEECo is paused, the robot only continues to its next waypoint or stops sooner if there seems to be an obstacle preventing its move and then it waits for the adaptation logic to instruct it further.

[17] http://wiki.ros.org/rosbag
[18] https://eclipse.org/

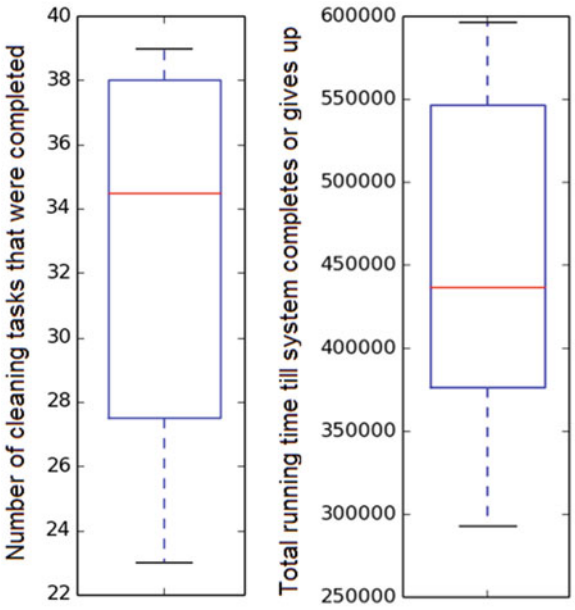

Fig. 7.10 Boxplots of results from 10 experiment runs

7.3.4 Obtaining Results

The testbed comes with a script which computes statistics of the evaluation from the logs collected in multiple simulation runs. It generates boxplots of the results for the last two metrics defined in Sect. 7.2.2 (as in Fig. 7.10).

7.3.5 Technical Architecture

Figure 7.11 shows the architecture of the testbed. Technically, it is a merger of four main existing modules. The contribution of the testbed lies in properly configuring them and bridging them by glue and synchronization code. The modules are:

- ROS Core – this module provides publish/subscribe middleware for robotic systems and the basic software of the robot. In particular, it implements the AMCL localization, navigation, and low-level movement control of the robot. The messaging system is used to interconnect robot basic software as well as to connect remaining modules described later.
- OMNeT++[19] – it is a network simulator. It runs independently of ROS. We have implemented a bridge between ROS Core publish/subscribe mechanism

[19] http://omnetpp.org/

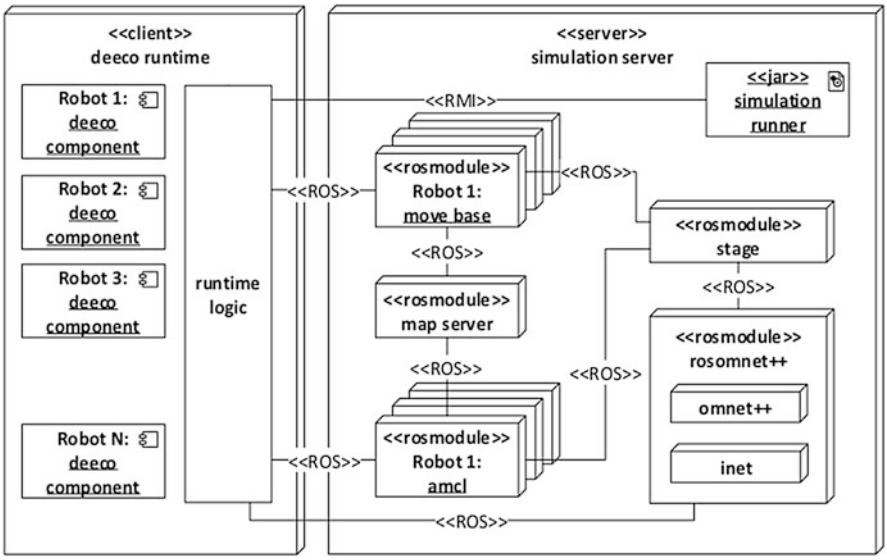

Fig. 7.11 Testbed deployment diagram

and OMNeT++, which exposes the MANET transceiver as a ROS topic. This allows modules connected to ROS to communicate. OMNeT++ simulates the latency, physical range, and interference of the communication based on robots' positions.

- Stage[20] – it is a robot simulator, which controls the simulation. It connects to ROS Core and simulates sensors and actuators of the robot given the simulated robot position and the map of the environment. Robots sensors and actuators are exported as ROS topics. The interface of simulated robot is the same as the interface of the real TurtleBot. The only difference is the usage of namespaces which enable deployment of multiple simulated robots into one ROS system.
- JDEECo – it provides the component abstraction and concepts for decentralized coordination as described in Sect. 7.3.1. It abstracts ROS topics on location and navigation and exposes them to DEECo components to allow for adaptation. It further exploits the ROS topic on MANET-based communication (backed by OMNeT++) to implement inter-component communication via ensembles. JDEECo again runs independently of ROS and is synchronized with it by a bridge that we have developed as part of the testbed.

[20] http://playerstage.sourceforge.net/

7.4 Evaluation

7.4.1 Example Adaptation Logic

We complement the model problem specification and the testbed with an example adaptation logic as part of the model problem. It provides a comprehensive example of the modeling concepts (described in Sect. 7.3.1) and also serves as evaluation of the testbed to perform simulation of physical, mobility, networking, and coordination concerns.

In the example adaptation, we tackle the problems described in Sect. 7.2.1 in the following way:

1. We introduce a process (on each robot), which periodically detects the situation when a robot is stuck. This is done by checking whether the robot is moving and whether the robot has a destination set. The robot that is not moving and wants to move is considered stuck.
2. If a robot is detected to be stuck, we select a random location from its queue of destinations and set it as its current one. This resets the navigation module in the robot and typically gets the robot to move. We monitor the outcome via the process described in (1) and repeat if no visible outcome is detected.
3. If another robot is stuck in close proximity (up to 1.5 m), we establish an ensemble with it. Within the ensemble, one robot adopts the current destination of the other robot and vice versa. This solves the (deadlock) situations when two robots meet in the office entrance and cannot proceed.

Strategy (3) is illustrated in Fig. 7.12. Ensemble membership is defined on lines 7–13, and destination adoption is defined on lines 20–24.

7.4.2 Lessons Learned and Limitations of the Testbed

The experience with development of the testbed on top of ROS led us to several observations, which we believe are of general interest. We thus share them here.

Generally, a relatively big surprise was the overall immaturity of the frameworks. This most likely stems from the fact that ROS is primarily used as a platform for controlling a single robot at real time. Though it has very flexible architecture, which allows running multiple robots within a single ROS system and allows connecting different environment simulators (e.g., Stage), the practice shows that these setups work out of the box only for trivial examples. Deploying multiple robots without careful configuration of the environment would make ROS or the Stage simulator crash. Similar story applies for OMNeT++, which is a mature and production-ready network simulator used in many applications. Nevertheless, when it comes to complex exercising of the mobile ad hoc network, the simulator again becomes very fragile, and without careful configuration and patching, it crashes for no obvious reason. From this perspective, we believe that even without the DEECo abstraction

```
1. @Ensemble
2. @PeriodicScheduling(period = 3000)
3. public class DestinationAdoptionEnsemble {
4.   double MAX_DIST_M = 3.0;
5.   long BACKOFF_MS = 10000;
6.
7.   @Membership
8.   bool membership(@In("coord.id") String
    coordId, ...) {
9.     return coordState == Block &&
10.            mbrState == Block &&
11.            !coordId.equals(mbrId) &&
12.        coordPos.distTo(mbrPos) < MAX_DIST_M;
13.   }
14.
15.   @KnowledgeExchange
16.   void exchng(@InOut("mbr.destination") destination, ...) {
17.     if (now-lastAdoption.value) < BACKOFF_MS) {
18.       return;
19.     }
20.     mbrAdoptedDestinations.value.add(coordDestination);
21.     mbrDestination.value = coordDestination;
22.     mbrRoute.value.add(coordDestination);
23.     mbrBlockCnt.value = 0l;
24.     lastAdoption.value = now;
25.   }
26. }
```

Fig. 7.12 Excerpt from example ACRC adaptation strategy

layer, the pre-configured testbed we provide can save a couple of months of painful debugging.

Another class of problems comes from the fact that though ROS has been used in simulations, it is not a discrete event simulator. It consists of a number of modules, which just run in wall-clock time. This means that (1) the simulation is non-deterministic and (2), if extra care is not taken, the system crashes because the simulator, ROS, OMNeT++, and DEECo are not synchronized. We have solved this problem by introducing explicit synchronization at critical places, but still one has to keep in mind that this solution does not result in fully deterministic simulations.

Surprisingly enough, our experience with developing the sample adaptation logic has shown that the wall-clock-timed simulation has certain advantages over a standard off-line discrete-event simulation. Since the system is live (and behaves as if the robots were moving in real time), one can watch the system as it runs, inspect the laser scans, etc. Additionally, it is possible to modify the system while it is running – e.g., a robot can be dragged by mouse to another location. While this is not important in classical batch simulations which focus on statistical comparison of different algorithms, it is very useful in debugging and especially in prototyping (which in fact is one of the primary goals of our testbed and the reason why we equipped the testbed with DEECo abstractions).

7.5 Testbed Structure

The complete testbed is available at http://d3s.mff.cuni.cz/projects/components_and_services/deeco/files/deeco-adaptation-testbed.zip. It contains the source code of the testbed, together with installation and usage instructions. Moreover, a pre-configured virtual machine image is included in order to enable rapid hands-on experience without the hassle of installing tons of libraries.

7.6 Related Work

In this section, we briefly review three model problems/exemplars that have been contributed to the self-adaptive systems community repository [8]. This is an ongoing effort to provide benchmarks to evaluate new techniques against the state of the art, a popular strategy in other communities such as performance engineering (DaCapo suite [1], SPEC benchmarks [11], Cloud Efficiency Metric [10]).

The automated traffic routing problem (ATRP) [14] is a model problem that can be used as benchmark for the evaluation of different self-adaptation mechanisms. ATPR features cars traveling on a map. Each car has a specific starting point, a specific destination, and a specific starting time. Each car has the goal to reach its destination by traversing the map while respecting the speed limits on the streets. Problems arise due to conflicts between individual goals (e.g., all cars select the same street resulting in traffic congestion on the street), traffic accidents, and road closures. ATRP can have solutions that are fundamentally different ranging from centralized to completely decentralized ones and generating answers that are optimal or suboptimal. These solutions can be compared w.r.t. dimensions such as scalability, answer quality, robustness to sensor uncertainty, etc.

To evaluate and compare ATRP solutions, ADASIM has been proposed [14]. ADASIM is a Java-based discrete-event simulator that simulates the execution of an ATRP solution on an ATRP instance. It provides configuration files for specifying the problem instance, built-in routing algorithms, traffic delay functions, filters for introducing measurement uncertainty, and Java interfaces that can be implemented to specify an ATPR solution. An event logging and analysis infrastructure is also provided. In summary, ATRP provides a vehicle suitable for experimentation with different self-adaptation strategies that try to resolve conflicts between goals of individual agents, prioritize between nonfunctional properties, and provide robustness to faults. Similarly, our model problem and testbed stand as a benchmark for self-adaptation mechanisms but focus more on run-time uncertainty and unreliable communication and coordination in sCPS.

Znn.com [3] is another model problem for self-adaptation. Znn.com is a news service that serves multimedia content to its customers. It is realized by a classical N-tier client-server architecture. The objective of Znn.com is to serve content while optimizing operating costs and keeping the response time bounded. Self-adaptation

capabilities are needed in order for Znn.com to react to spikes on user load or other external changes while satisfying its objectives. In such cases, Znn.com can choose from a limited number of predesigned adaptation decisions, e.g., switching the content served from multimedia to textual or incrementing the server pool size. While Znn.com is primarily suitable for comparing self-optimization solutions, our model problem and exemplar are more suitable for comparing solutions that focus on self-healing and survivability (robot unblocking, deadlock resolution) properties of sCPS.

Tele Assistance System (TAS) [12] is an exemplar for self-adaptation in the area of service-based systems. TAS aims to aid patient suffering from chronic conditions via tele-assistance. It is realized by wearable sensors measuring vital parameters and three remote services for data analysis, medication delivery, and ambulance dispatching in case of emergency. TAS comes with a number of generic adaptation scenarios. Each scenario consists of the type of uncertainty that warrants self-adaptation (e.g., service failure), appropriate self-adaptation actions (e.g., switching to equivalent service), and type of quality attributes (QoS) impacted (e.g., cost). For measuring the satisfaction level of each QoS, respective metrics are specified. A reference implementation of TAS [12] provides a convenient way of comparing self-adaptation solutions w.r.t. user-specified requirements in user-specified settings (instances of the generic TAS scenarios) by simulating them and analyzing the results with built-in graphical tools. While an excellent representative exemplar for self-adaptive systems, TAS focuses specifically on service-based systems, not CPS.

7.7 Summary

Responding to the pressing need for model problems and testbeds to evaluate the research ideas in the area of self-adaptive smart cyber-physical systems (sCPS), we have presented ACRC, a model problem in the realm of sCPS that lends itself to a number of self-adaptation techniques that increase its self-healing, survivability, and self-optimization properties. It facilitates the process of trying out and comparing self-adaptation solutions to this problem. Our pre-configured testbed provides a starting point for experimental research. Moreover, the experiments can be easily extended to actual robots as the simulation shares interface with off-the-shelf robotic platform. We hope that ACRC will help increase the awareness of the yet-to-be-addressed challenges in the exciting field of self-adaptive sCPS and drive further advances in the field.

Acknowledgements The work on this paper has been partially supported by the Charles University Grant Agency project No. 391115 and Charles University institutional funding SVV-2016-260331. This work is part of the TUM Living Lab Connected Mobility (TUM LLCM) project and has been funded by the Bayerisches Staatsministerium für Wirtschaft und Medien, Energie und Technologie (StMWi).

References

1. Blackburn, S.M., Garner, R., Hoffmann, C., Khang, A.M., McKinley, K.S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S.Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J.E.B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., Wiedermann, B.: The dacapo benchmarks: java benchmarking development and analysis. SIGPLAN Not. **41**(10), 169–190 (2006). https://doi.org/10.1145/1167515.1167488
2. Bures, T., Gerostathopoulos, I., Hnetynka, P., Keznikl, J., Kit, M., Plasil, F.: Deeco: an ensemble-based component system. In: Proceedings of the 16th International ACM Sigsoft Symposium on Component-Based Software Engineering, CBSE'13, pp. 81–90. ACM, New York (2013). https://doi.org/10.1145/2465449.2465462
3. Cheng, S.W., Schmerl, B.: Znn model problem. http://self-adaptive.org/exemplars/model-problem-znn-com. Accessed 22 Jan 2016
4. Kim, K., Kumar, P.R.: Cyber-physical systems: a perspective at the centennial. Proc. IEEE **100**(Centennial-Issue), 1287–1308 (2012). https://doi.org/10.1109/JPROC.2012.2189792
5. Matena, V., Bures, T., Gerostathopoulos, I., Hnetynka, P.: Model problem and testbed for experiments with adaptation in smart cyber-physical systems. In: Proceedings of the 11th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS'16, pp. 82–88. ACM, New York (2016). https://doi.org/10.1145/2897053.2897065
6. NIST: Cyber physical systems: situation analysis of current trends, technologies, and challenges. In: NIST CPS Workshop (2012). http://events.energetics.com/NIST-CPSWorkshop/pdfs/CPS_Situation_Analysis.pdf
7. Salehie, M., Tahvildari, L.: Self-adaptive software: landscape and research challenges. ACM Trans. Auton. Adapt. Syst. **4**(2), 14:1–14:42 (2009). https://doi.org/10.1145/1516533.1516538
8. Self-adaptive systems community repository. http://self-adaptive.org/exemplars. Accessed 22 Jan 2016
9. Sha, L., Gopalakrishnan, S., Liu, X., Wang, Q.: Cyber-physical systems: a new frontier. In: Singhal, M., Serugendo, G.D.M., Tsai, J.J.P., Lee, W., Römer, K., Tseng, Y., Hsiao, H.C.W. (eds.) IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing (SUTC 2008), Taichung, 11–13 June 2008, pp. 1–9. IEEE Computer Society (2008). https://doi.org/10.1109/SUTC.2008.85
10. Shtern, M., Smit, M., Simmons, B., Litoiu, M.: A runtime cloud efficiency software quality metric. In: Companion Proceedings of the 36th International Conference on Software Engineering, ICSE Companion 2014, pp. 416–419. ACM, New York (2014). https://doi.org/10.1145/2591062.2591127
11. Spec benchmarks. http://www.spec.org/benchmarks.html. Accessed 22 Jan 2016
12. Weyns, D., Calinescu, R.: Tele assistance: a self-adaptive service-based system examplar. In: Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS'15, pp. 88–92. IEEE Press, Piscataway (2015). http://dl.acm.org/citation.cfm?id=2821357.2821373
13. Weyns, D., Malek, S., Andersson, J.: Forms: a formal reference model for self-adaptation. In: Proceedings of the 7th International Conference on Autonomic Computing, ICAC'10, pp. 205–214. ACM, New York (2010). https://doi.org/10.1145/1809049.1809078
14. Wuttke, J., Brun, Y., Gorla, A., Ramaswamy, J.: Traffic routing for evaluating self-adaptation. In: Müller, H.A., Baresi, L. (eds.) 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS, Zurich, 4–5 June 2012, pp. 27–32. IEEE Computer Society (2012). https://doi.org/10.1109/SEAMS.2012.6224388